EXHIBIT "C"

Cluster Security: Secured Execution Environment System Design"
dated April 16, 2001

Cluster Security: Secured Execution Environment System Design

Version 1.1

Monday, April 16, 2001

Clusters Software and SP System Services Department

Design Owner:
Bob Gensler / bobgens@us.ibm.com

Authors:
Bob Gensler / bobgens@us.ibm.com
Serban Maerean / serban@us.ibm.com
Larry Parker / cparker@us.ibm.com
Hemant Suri / hemant@us.ibm.com

IBM Corporation
RS/6000 SP/Clusters Development
Poughkeepsie, N.Y.

Online version is the master.

1.0 Abstract/Line Item Overview

This document describes the infrastructure and interfaces provided for Linux and AIX cluster environments to provide a secured application execution environment. The infrastructure and interfaces are designed to provide these capabilities:

- Provide a generic interface for authentication and authorization, to make it easier for cluster components to implement security, and to permit cluster components to use common code for authentication and authorization regardless of the specific security mechanism used to provide security within the cluster.
- Provide the capability for system administrators to install and configure an AIX or Linux cluster to use an authentication method that the customer can select. The initial offering will support Kerberos 5 based authentication, Unix operating system based authentication, and permit the addition of other authentication mechanisms in the future. Utilities are provided for setting the authentication methods, providing the necessary information about these methods, and enable these methods within the cluster.
- Provide the ability to easily port the infrastructure to the SP computing environments. This permits existing SP subsystems to migrate towards a common security interface in the future, and can facilitate the porting of existing SP components to Linux and AIX clusters. Common source code can be achieved for components whether the code is intended for the SP, AIX clusters, or Linux clusters.
- Provide equivalent security functionality and interfaces for 32-bit computing environments and 64-bit computing environments.

2.1 Line Item Tracking Information

This system design is associated with Pipeline Line Item AU7 and CMVC line item tracking feature 69772.

1. "aznAPI" is a proposal offered by DASCOM Incorporated, 1999.

2.2 Implementation Staging Information

This document does not discuss the delivery schedules or staging of the function proposed by this design. Those interested in this information are directed to consult the release plan for this line item. This information is available from the Security manager and the Security team leader.

3.0 Requirements

3.1 Requirement As Provided

The following is a list of requirements which can apply to a security infrastructure offering for Linux and AIX clusters. Those requirements accepted by this specific design are listed in following sections. Requirements not addressed or postponed will be explained as well.

1. Provide an infrastructure and a corresponding set of interfaces for application use to provide a secured operating environment in Linux and AIX clusters. Provide facilities for Linux and AIX cluster components to authenticate and authorize other parties. Provide functions that permit Linux and AIX cluster components to transmit information securely, in a fashion that permits the component to verify who transmitted the data and that protects the privacy of the data. Provide storage facilities for the data objects necessary to authenticate and authorize other parties.
2. Provide an interface that promotes the development of common security code within Linux and AIX cluster components. Remove the need for Linux and AIX cluster components to understand the specifics of the security mechanisms in use within the cluster, and reduce the need for each component to implement common functions such as error handling.
3. Provide a security infrastructure that does not require a specific security mechanism to be used in the customer's environment. Allow the customer the flexibility to use the security mechanism that the customer prefers. Do not force the customer to make use of a specific mechanism. An example of this requirement would be to not force a customer to install and configure Kerberos5 if another security mechanism is preferred by the customer.
4. Provide an authentication scheme that exploits existing operating system assurances that the user is authentic. These assurances are weaker than assurances provided by a "trusted" third party security mechanism, but can be used to provide minimal (but certainly not foolproof) security assurance when it is not possible for the customer to install or use a "trusted" third party security mechanism such as Kerberos version 5. An example of such an authentication mechanism is the Unix domain socket (UDS) authentication scheme that is used extensively within PSSP and RSCT component that are being provided in the initial Linux and AIX cluster. Another example is the host-based trust employed by remote commands such as **rsh** and **rcp**.
5. Provide a security infrastructure capable of supporting security mechanisms that may become available in the Linux and AIX cluster environment in the future. Providing this support should only require a configuration alteration of the security infrastructure, and not an alteration of any Linux or AIX cluster component binaries or source code.
6. Provide an infrastructure and interface compatible to previously provided security infrastructures on other supported AIX platforms, such as PSSP's S3 security services subsystem, to assist in porting existing PSSP components to Linux and AIX clusters.
7. Provide an infrastructure and interface that can support nodes existing within Linux clusters and other AIX clusters or SP configuration at the same time. The infrastructure and interfaces would need to coexist and possibly cooperate with existing security infrastructures previously supplied

for these platforms, such as PSSP's S3 security services subsystem. Such a solution would facilitate cooperation and workload sharing between Linux clusters and existing AIX clusters or SP systems.

8. Provide a secured environment for Linux and AIX cluster software installation, to prevent the installation of malevolent software or utilities that can jeopardize system security and function.
9. Provide support of security mechanisms that are consistent with the long-term direction of Linux and AIX development within IBM. This direction currently dictates that Kerberos version 5 be used as the security mechanism for AIX and Linux offerings initially, with support for certificates based authorization following in the near future.
10. Allow for an AIX or Linux cluster to be used in an insecure mode. Allow a customer to configure a cluster where no security software such as Kerberos version 5 is required to operate the system. Do not require the installation of security software if a customer chooses not to secure the cluster, and do not impose extra overhead on the cluster operation in these modes.

Linux cluster component development places additional generalized requirements upon a software development project:

1. Do not require the use of a globally accessible, distributed data repository. Unlike the SP's System Data Repository, Linux and AIX clusters do not provide a default distributed repository. Any design that requires a distributed repository must therefore furnish one.
2. Do not require a single point of control. Allow system administration tasks to be performed from any node within the cluster. Do not serialize specific functions to a node based on the node's cluster identity. Unlike the SP's Control Workstation, Linux and AIX clusters cannot be assumed to provide a centralized point of control for administration and coordination purposes.

The following are a set of general requirements imposed upon any software project developed by the IBM Unix Development Lab.

1. Provide a reliable security infrastructure and interface. Reliability is defined as the assurance that the infrastructure and interfaces will always function properly when used correctly, and when all necessary resources are available. In other words, the product is to be "bug free".
2. Provide interfaces that support 32-bit and 64-bit execution environments, without restricting the function offered to clients in either environment.
3. Provide a security infrastructure and interface that can easily be ported from the Linux cluster to AIX clusters and SP environments. This is consistent with the laboratory's long term strategy of merging Linux cluster and AIX cluster offerings into a common spectrum of offerings for all Unix based platforms.
4. Provide a solution that is scalable to Linux and AIX clusters with numerous nodes. Use of the security infrastructure and interfaces in a large Linux or AIX cluster should not impose more operational overhead than the use of the same infrastructure and interfaces in a smaller Linux or AIX cluster.

5. Provide a solution that achieves the correct balance between security of the system and performance impacts on users of the security infrastructure. These tend to be contrary goals.
6. Provide sufficient serviceability support within the product. Provide sufficient ability to troubleshoot the security infrastructure during execution, and to diagnose potential problems in the infrastructure.
7. Provide interfaces that are consistent in look-and-feel with industry standard interfaces. Follow existing internal conventions and industry standards for such interfaces wherever possible. Consider the usability of such interfaces in the design and implementation.

3.2 Requirements As Interpreted

A security infrastructure must be provided for Linux and AIX cluster computing environments. This infrastructure must provide function to cluster services, client applications, and client users to *authenticate* other parties (ensure that the other party is who the other party claims to be) and to ensure that other parties are *authorized* to request or perform specific functions (ensure that the other party has sufficient privilege to carry out the request it is making). The security infrastructure must provide clear assurance that the identities of cluster services, client applications, and client users cannot be forged by malevolent parties, which would permit a breach of system security.

Several technologies exists to provide authentication today in a Linux environment, just as in an AIX cluster environment. Linux customers are free to choose the software they deem best -- Kerberos5, DCE, Verisign -- for their purpose. Unlike AIX cluster and SP environments where IBM provides the total computing solution, IBM provides only portions of the Linux cluster solution. IBM Linux cluster software can be installed on a wide variety of Linux platforms. This poses a challenge to IBM Linux cluster software: the software must be able to operate correctly in a wide variety of configurations. Security is one area where a variety of configurations may exist, and given that IBM cannot control the variety of configurations, IBM Linux and AIX cluster software needs to be flexible enough to operate in a variety of security environments.

This flexibility can be achieved in a number of ways. One method is to make the cluster software aware of all possible security methods that can be employed. The software would detect which method was currently in use, and employ the correct source code to utilize those methods. Unfortunately, this approach requires each cluster component to invent their own means for dealing with multiple security methods, leading (at best) to redundant code or (at worst) to faulty code employed in some cluster components. This approach also requires a reworking of all cluster software components whenever new security mechanisms are made available for the cluster platform to support the new mechanism, requiring new tests of the same software components.

To avoid these problems, an abstraction layer is inserted between the cluster software components and the underlying security mechanisms used in the Linux or AIX cluster. This abstraction layer is responsible for determining what security methods are available in the cluster, and employs the appropriate code to perform the necessary security functions on behalf of the cluster component.

Common error checking and handling functions can also be integrated into this abstraction layer, helping to achieve further code commonality. The cluster component remains unaware of the underlying security methods. Should a new security mechanism be made available for the cluster, only this abstraction layer needs to be modified to support the new method, and only the abstraction layer would need to be tested to verify that the support is correctly implemented. cluster software components, abstracted from the underlying mechanism, require no modifications or new testing.

Injecting this new abstraction layer helps the cluster software components to implement common code for performing security functions. However, achieving common source code is insufficient. The security infrastructure must provide the capability for Linux and AIX cluster components to employ the same executable program in the cluster, instead of employing a different executable for each possible security configuration. In addition to providing a common interface for cluster services and client applications to use, the security infrastructure must provide a common service for these services and applications to use. The service or application should execute the same binary every time in the same cluster platform. The security infrastructure must be capable of determining what underlying libraries or modules would be needed to interface with the configured security mechanisms, and load those modules as the service or application executes. This function is abstracted from the cluster components, who are not aware that it is being done on their behalf by the security infrastructure.

Another goal of achieving code commonality is to develop common source code for all platforms where the software component is offered. Using common source code for all platforms reduces the development and servicing complexity of the software. Injecting an abstraction layer to handle mechanism differences helps to achieve code commonality on one specific platform only. Components provided for multiple platforms -- AIX clusters, SP systems, and Linux clusters -- do not derive the full benefit of common code from such an abstraction layer, unless that abstraction layer is available on all these platforms. Without it, the software components still need to provide separate code paths for different platforms. Although this design assumes that the initial implementation will be provided for Linux clusters, the infrastructure and interfaces provided by this design are designed to be easily ported to AIX and other Unix based platforms. Industry standard coding practices and interfaces are used wherever possible, and platform specific code is isolated to specific code paths to allow for easy future porting efforts.

The security infrastructure must be capable of supporting multiple security mechanisms. Because IBM does not have control over the entire Linux cluster software solution, the security infrastructure needs to be flexible enough to support a number of possible security mechanisms. Initial releases of the Linux cluster security infrastructure will restrict what security mechanisms can be employed in the initial offering, but the security infrastructure will be designed so that support of additional security mechanisms can be provided in later releases through provision of mechanism-specific modules, instead of requiring a reworking of the security infrastructure itself. This reduces the retesting effort of the security infrastructure to a verification of the new mechanism-specific modules, instead of requiring a complete new test of the entire security infrastructure source code.

In addition to authentication, cluster components require the capability of authorizing other parties. Even if a party has been authenticated by a cluster component, it may choose to restrict certain resources or functions from all but a specific set of requestors. For this reason, cluster component require function to determine which parties are authorized and which are not.

As with authentication, there are several existing methods in Linux clusters for providing this authorization function. Once again, IBM cannot control what specific mechanism is used to provide this capability. The cluster components either need to know about all possible mechanisms that can be used, or this information also needs to be abstracted from the cluster components just as the authentication mechanisms have been abstracted. The wiser choice is to abstract the mechanism and to provide a common interface for all cluster components to use for authorization purposes. These interfaces allow cluster components to create, modify, and remove authorization privileges for specific parties, and to verify whether specific parties are granted sufficient privileges within this record for the Linux cluster component to allow that party to proceed.

The security infrastructure provides interfaces for 32-bit and 64-bit execution environments. The same function is available to clients in both environments. Application programming interfaces follow conventions specified by POSIX and existing security interfaces in the industry; should these conventions differ, the interfaces provided by this design choose to imitate existing security interfaces to provide a consistent look-and-feel to them. Interfaces are also provided to security system administrators to configure the security infrastructure.

Performance remains a key requirement in cluster environments. Clusters are created to improve the performance of large workloads by segmenting and sharing the workload amongst many processors. Therefore, any new infrastructure introduced to a cluster environment must avoid degrading the overall performance of the cluster or a single node in that cluster any more than absolutely necessary. Function should be performed in the shortest code path possible, with the least amount of blocking and waiting necessary. Unfortunately, the requirement to secure the cluster operation is contrary to these goals of performance. Providing a secured environment requires that applications incorporate some degree of paranoia. Applications must verify identities and permissions of their clients, adding to the code path length. Such verifications may require blocking or waiting for a "trusted" third party confirmation, further adding to the operational overhead.

To resolve the inherent conflict between the performance and security requirements, a trade-off must be made. Some degree of operational overhead is imposed upon users of the security infrastructure. The security infrastructure and its interfaces are designed to minimize the amount of overhead necessary to provide a secured operational environment. Users of the infrastructure and interfaces must accept this overhead as a consequence of using a secured execution environment. Should this level of overhead be deemed unacceptable, the application will need to make its own trade-off between security and performance, and decide whether the performance needs outweigh the needs of security for their own product.

Requirements

The security infrastructure and interfaces provided by this design are scalable to clusters containing numerous nodes. Applications using the infrastructure should not incur more operational overhead from the infrastructure itself if the application executes on a 128 node Linux or AIX cluster than it would executing on a 16 node cluster. The security infrastructure avoids use of global data requiring exclusive access from a single node, and any cases where such access is required is restricted to configuration activities performed by security administrators only.

This design introduces a software layer on Linux and AIX clusters which is critical to the proper execution of cluster components. Because of this, the security infrastructure provides reliability assurance and serviceability features above and beyond those provided by typical Linux software. Compliance to the IBM Unix Development Lab's internal policies of thread safety and reliability is expected. Cluster components must be assured through sufficient code examination and testing that the security infrastructure and related interfaces are reliable. The infrastructure also provides serviceability features such as execution traces and diagnostic utilities to assist in the run-time analysis and troubleshooting of the infrastructure. When properly installed and invoked, the security infrastructure will provide a minimum level of execution tracing that can be used to verify the infrastructure's correct operation, plus additional levels that can be activated at the system administrator's request.

This additional layer of software is included as part of the core set of utilities required by the Linux and AIX cluster software. The security infrastructure is installed even in cases where the customer may wish to operate an insecure cluster. This is required because all cluster trusted services will be coded to invoke the security infrastructure routines in all cases, whether or not the cluster is configured to be secure. Instead of forcing the cluster trusted services to check whether the cluster is operating in a secured mode, the security infrastructure makes this determination. This provides a consistent means for determining the security status of the cluster, permits the cluster trusted services to use the same source code path for both secured and unsecured clusters, and permits the cluster trusted services to handle a shift from unsecured to secured operation more efficiently. Operating any Linux or AIX cluster in an insecure mode is not likely, but should such a cluster be configured, a minimal amount of overhead is imposed on the cluster trusted services. The security infrastructure will detect that the cluster is executing in an unsecured mode, and label all users as unauthenticated.

4.0 System Design Overview

The goal of this system design is to provide a secured execution environment for Linux and AIX cluster software components that reduces the complexity of interacting with the security infrastructure and promotes the development of common code for these components.

Specific objectives of this design are:

- Provide trusted services, client applications, and system users with an identity that can be used to verify them and their access permissions. These identities are provided through the use of existing industry security mechanisms. Kerberos version 5 is chosen as the security mechanism for the initial Linux and AIX cluster release. Support the addition of alternate security mechanisms for this purpose in the future.
- Provide a generic, security mechanism independent authentication and authorization interface to trusted services and application clients. This interface abstracts the underlying security mechanism, promoting the use of common code in the cluster components and applications. A new application programming library is added to the cluster software offering to achieve this goal.
- Provide a generic, security mechanism independent interface to trusted services and application clients to transmit information in a secured manner. This interface must be capable of protecting sensitive data so that only authentic parties can interpret the data, and also capable of signing the data to prove the authenticity of the transmitter. The same application programming library proposed in the previous bullet also addresses this objective.
- Provide methods for installing the security infrastructure and the generic abstraction layer interface. CtSec supports the use of the installation mechanisms employed by IBM Linux cluster software, as well as the **installp** method used in AIX clusters..
- Provide methods for configuring the security infrastructure. A combination of manual procedures and semi-manual convenience utilities are provided to perform this function.
- Provide a complete set of administration commands to control the operation of the security infrastructure.

These new functions are provided in a manner that does not prevent existing cluster components and applications from making use of existing security services and interfaces, such as using Kerberos5 directly.

This design makes the simplifying assumption that all nodes within a cluster make use of the same underlying security mechanisms. If any node does not make use of the same security mechanisms (for example, the node chooses to use Certificates based authorization only, while other nodes in the cluster support both Certificates and Kerberos 5), the node is considered *not* to be a member of the cluster. In those cases, the security infrastructure will not guarantee that parties from that node can be authenticated or authorized. Another way of stating this is that a cluster is (partially) defined as a set

of nodes using a common set of security mechanisms; any node not meeting that criterion is not considered a member of the cluster.

4.1 Background - Current Linux Cluster Security Environments

This design specifies an infrastructure and related set of interfaces to provide a secured execution environment for Linux cluster software. A description of the current Linux cluster security environment is necessary, to understand how the new offering fits in with the existing methodology.

Traditionally, Unix applications that have required a secured execution environment have needed to select one or more security mechanisms to provide security identities to various parties in the system: trusted services, client applications, and system users. Kerberos version 4, Kerberos version 5, and DCE are some of the mechanisms that have been traditionally employed in this capacity. These mechanisms provided unique identities to all parties, and acted as a broker on behalf of these parties during the setup of a secured communication path. The parties involved in the transaction would trust the security mechanism to verify the identity of the other participant, ensuring that the other participant is valid and not trying to forge another identity to gain access to restricted aspects of the system.

Using a specific security mechanism required the application to use mechanism specific interfaces for some security functions. While industry convention insured that a security mechanism provider would provide a GSSAPI library for authentication functions, applications had to resort to mechanism specific interfaces to obtain their own security identity: to "log on" to the system and obtain their security credentials. Applications seeking to support a variety of differing security mechanisms were forced to provide mechanism specific code for obtaining their security identity, which differed depending on the security mechanism that was in use on the system. Either the application would have to provide different binaries for each security mechanism it supported, or it would have to incorporate logic within it to detect which security mechanism was in use and to load a module specific to that mechanism.

The following diagram depicts a model in which an application attempts to support different security mechanisms within the same binary. Such approaches are necessary for service daemons that listen upon a network port, where only one executable program must be configured to accept requests on that port. To support multiple security mechanisms, the application partitions the security interaction code from the rest of the application. The application invents new routines to handle the security login function, which will eventually map to corresponding routines in the underlying security mechanism's library. At execution time, the application determines which mechanism specific module it must load, corresponding to the security mechanism that as been installed and configured on that system.

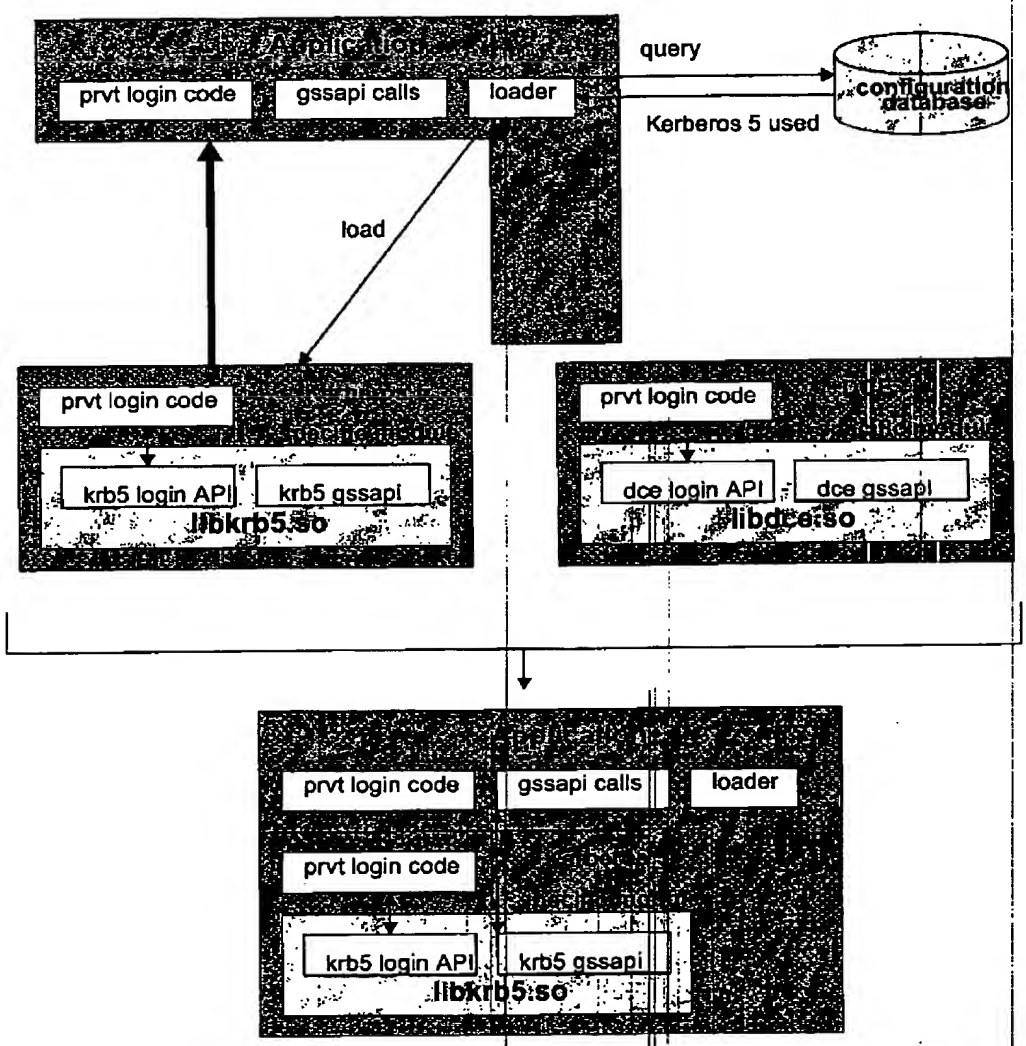

FIGURE 1 Example structure of an application that supports multiple security mechanisms in the same executable program, depicting the application's point of view.

Note that in this example, the application must provide multiple binaries, must determine what underlying security mechanism is currently in use, and load the correct binary for the correct security mechanism. The same procedure would have to be repeated for any application on the node that would attempt to support multiple security mechanisms.

A facility has been made available on Linux platforms to help simplify an application's efforts in "logging in" to the security mechanism and obtaining its credentials. PAM -- Pluggable Authentication Modules -- remove the need for Linux applications to write private "log in" code. PAM's purpose is to abstract the specifics of the underlying security mechanism's "log in" procedures, instead providing a common interface that can be used regardless of the security mechanism in use on the system. PAM handles the problem of determining which mechanism is in use, and translating the general "log in" call to the security mechanism specific version. Once the security identity and credentials are obtained via the PAM interface, the application makes use of the security mechanism's GSSAPI interface to authenticate other parties. Future versions of AIX have indicated an intent to provide PAM support in that platform as well.

While the PAM mechanism abstracts the "log in" and credential retrieval process for an application, the application must still know what underlying security mechanism is in use. The application needs to know what library must be used to perform authentication functions. Although the introduction of PAM reduces the amount of mechanism specific code that an application needs to develop, the application still needs to include logic for determining what underlying security mechanism is in use, and for selecting the correct library to use.

The following diagram depicts an application using PAM to abstract the security mechanism "log in" function. While a certain amount of mechanism abstraction is achieved using PAM, the application is not truly abstracted from the underlying security mechanism. Each application attempting to use PAM in achieving a secured execution environment must still provide logic for determining the underlying security mechanism and using the correct library for that mechanism.

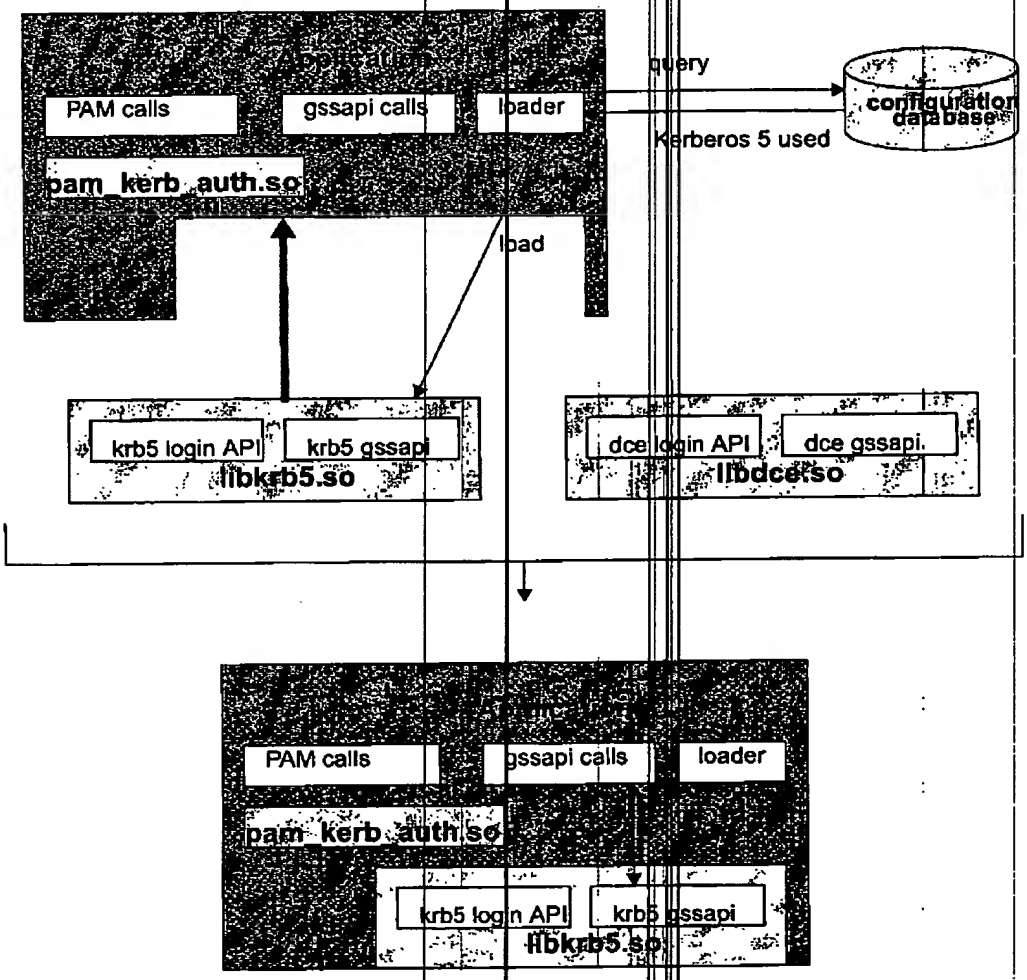

FIGURE 2 Example structure of an application that supports multiple security mechanisms, using the Pluggable Authentication Module (PAM) mechanism for obtaining security identity and credentials.

4.2 CtSec Mechanism Abstraction Layer - The Security Infrastructure

CtSec is the security infrastructure provided for Linux and AIX clusters by this design. The infrastructure implements a programming library callable by C and C++ applications that provides the following general categories of security function

- Retrieving the application's security identity and credentials ("log in" function)

- Authentication of the requestor
- Authorization of the requestor

From the application point of view, no knowledge of the underlying security mechanism is required. The application simply binds to the CtSec shared library. No decisions need to be made within the application as to what security mechanism is in use and what interfaces to load and call. These decisions are left to the CtSec library itself, and are performed invisibly in the application's point of view.

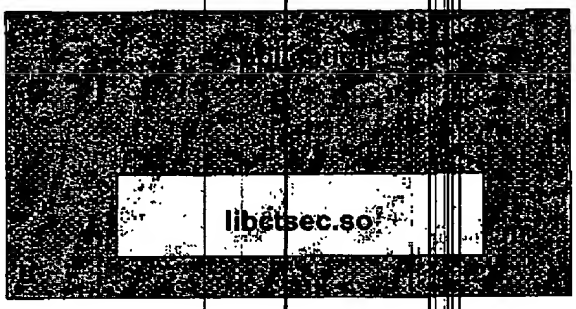

FIGURE 3 Example structure of an application using the CtSec library to support multiple security mechanisms, from the application point of view.

CtSec provides an interface independent of the underlying security mechanism for applications to obtain their security identity and credentials, authenticate other parties to establish a secured transaction environment, and verify the authorization of other parties in a secured transaction environment. CtSec maps these mechanism independent interface calls with the underlying mechanism specific interfaces to perform the function on behalf of the application. The internal components of CtSec are depicted in the following graphic.

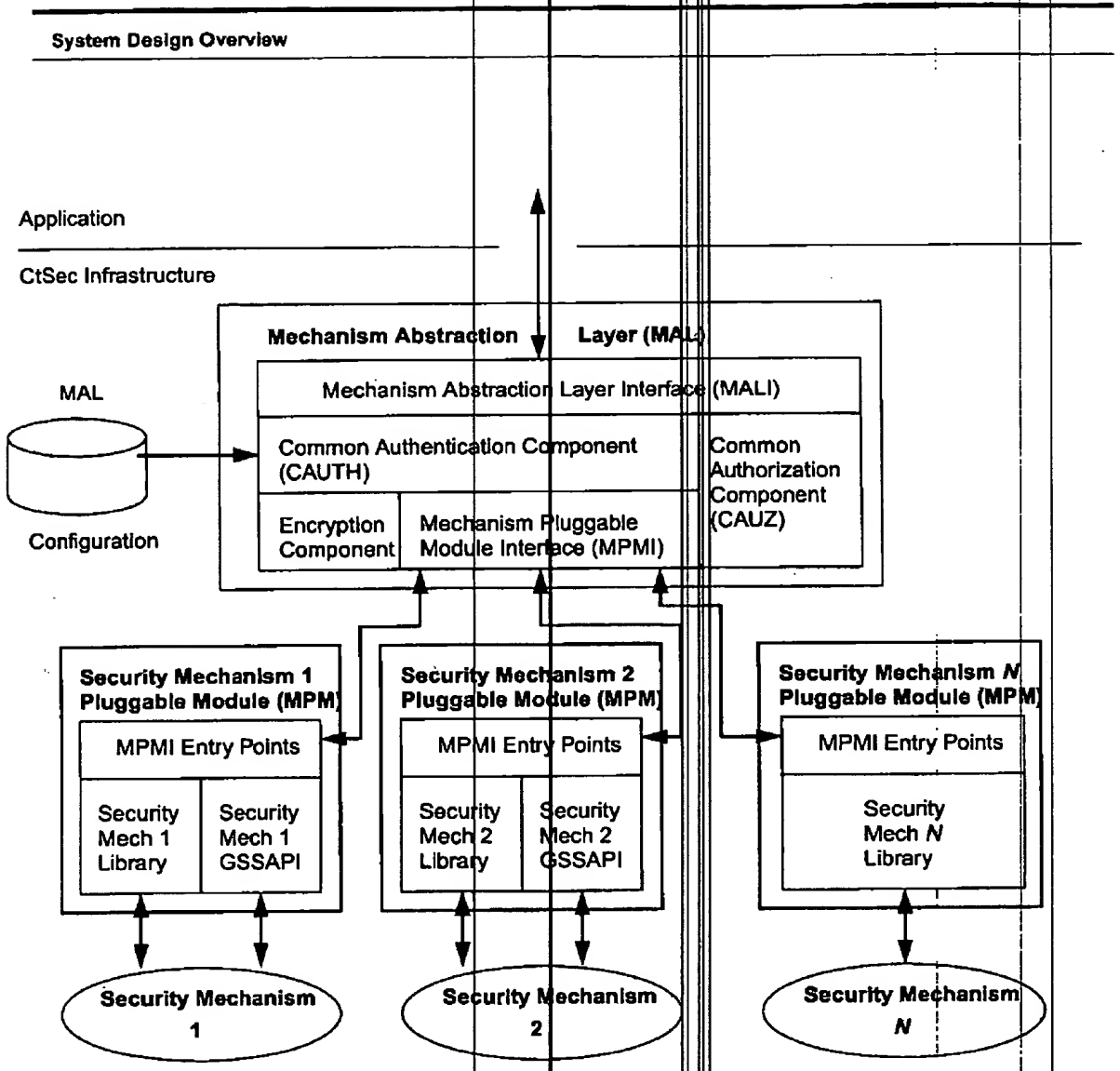

FIGURE 4 Internal components of the CtSec infrastructure.

Trusted services and client applications invoke the CtSec Mechanism Abstraction Layer Interface (abbreviated in this document as MALI) to perform security related functions. MALI interprets the application's request and translates it to the proper request for the next level of function. MALI supports three primary categories of security function:

- Obtaining the application's security identity, security credentials, or keys. This function is provided by the Common Authentication Component (abbreviated in this document as CAUTH).

- Authenticating other parties and processing secured data between authenticated parties. This function is also provided by the Common Authentication Layer (CAUTH).
- Verifying the authorization of other parties. This function is provided by the Common Authorization Layer (abbreviated as CAUZ in this document).

The MALI, CAUTH, and CAUZ layers collectively form the Mechanism Abstraction Layer (abbreviated MAL in this document). CtSec provides this layer, as well as the individual Mechanism Pluggable Modules (abbreviated as MPMs in this document) tailored to interface with specific underlying security mechanisms supported by CtSec. The Mechanism Pluggable Modules permit CtSec to obtain security mechanism specific identities and credentials for the application, using the interfaces that the security mechanism provides, such as GSSAPI libraries if the mechanism provides such support.

4.2.1 CtSec Common Authentication Layer (CAUTH) Component

CAUTH is responsible for obtaining a security identity, security credentials, and keys for an application by using the currently configured security mechanisms. CAUTH is also responsible for performing functions required for authentication of other parties, and for processing data for use between these parties in a secured execution environment. This component is requires an understanding of which security mechanisms are currently configured on the node, loading the correct pluggable module (MPM) to interface with those configured mechanisms, and obtaining the security identity and credentials for the application.

The CAUTH component is provided as part of the CtSec software, and consists of code developed within the laboratory. For CAUTH to function properly, the pluggable module (MPM) associated with the configured underlying mechanism must also be installed, and CAUTH must be informed about the MPM's location and any special instructions on how the module must be loaded.

4.2.1.1 CAUTH Configuration Information

CtSec's CAUTH component must be informed of the security mechanisms installed on the node, and the location of the mechanism pluggable modules (MPMs) associated with these security mechanisms. Without this information, CAUTH is unable to locate the appropriate module to load to perform authentication functions.

CAUTH configuration information is recorded to a private location. This location is expected to reside locally on all nodes in the cluster. The name and directory location of this file will be provided by the component design.

The CAUTH configuration can be implemented as a simple ASCII file, consisting of one record for each security mechanism that is installed on the node. The following information is required in each entry.

Mechanism Mnemonic — A symbolic name for the security mechanism. The mnemonic is a string such as "krb5", "krb4", "dce", "cert", etc.

Module Path	The fully qualified path name for the mechanism pluggable module (MPM) provided for the mechanism named in the mnemonic.
Invocation Options	Arguments used during the loading of the MPM, if special options are required.

CtSec configuration procedures permit the system administrator to submit this information without directly modifying the configuration file. Direct modification of this file, though possible, is discouraged. The purpose of the CtSec configuration procedures are to ensure that consistent information is provided for the cluster trusted services that are installed as part of IBM's cluster software.

4.2.2 CtSec Common Authorization Layer (CAUZ) Component

CAUZ provides applications with utilities to verify a party's authorization to access protected data or function within the application. CtSec's CAUZ component provides authorization capabilities that do not depend upon the authorization capabilities of the underlying security mechanisms. This direction is taken because not all security mechanisms currently available for AIX and Linux Clusters provide their own authorization capabilities. To ensure that CtSec is always capable of providing this capability even when the underlying mechanisms do not support, CtSec CAUZ provides its own.

In some security mechanisms, each instance of a trusted service is required to have an unique security identity. For example, the trusted service "zathras" on node "alpha" of a Linux cluster must have a different security identity than the instance of "zathras" running on node "beta", even though both instances comprise the same service. This poses a problem for administering the authorization rules for these trusted services. Any authorization mechanism protecting a resource or a function, such as an access control list (ACL), would have to contain one entry for each instance of a trusted service. The more trusted services, the longer the ACL must be; the more nodes within a cluster, the longer the ACL must be. Not only does this pose an administration inconvenience, it also poses a scaling problem for trusted services attempting to function in large Linux clusters.

Some security mechanisms offer security identity groups to assist with this problem. Security identity groups work in the same fashion as user groups work in Unix based systems. A security identity can be assigned to a group membership, and this membership can later be verified during authorization checks. Instead of needing to know the entire list of authorized identities, access can be granted based on group membership. Any identities that are members of a specific group are granted access, unless that user was explicitly denied access in another access control entry. Unlike security identities, the number of groups would not be required to increase as the size of the cluster increases; only the membership within already established groups would increase. Instead of creating an ACL based on security identities, ACLs can instead be created using groups. Instead of the ACLs growing in direct relation to the size of the cluster, the ACLs would stay the same size, helping to remove the scaling problem.

Unfortunately, not all security mechanisms provide security identity grouping[1]. To address this shortcoming, CtSec implements its own security identity grouping scheme, by mapping security identities

to local operating system identities, and associating these local operating system identities with operating system groups. On each node within the cluster, a local operating system identity is created that is mapped to an application's security identity. This local identity is then associated with one or more operating system groups, using standard Unix identity and group associations. When a service provider checks the authorization of a service requestor, it obtains the local operating system identity associated with the requestor's network identity, and from that obtains the list of groups associated with that identity from the operating system. The service provider then scans the access control list for matches using the following order:

- Searches for matches to the requestor's network identity
- Searches for matches to the requestor's mapped local operating system identity
- Searches for matches to the requestor's local operating system group membership

The mapping can be provided either by the underlying security mechanism, or by mapping software provided by CtSec or the Cluster software. The choice on which to use is left to the component design to decide.

To demonstrate, consider this example of a service provider -- "greatmachine" on node "epsilon3" -- which employs access control to grant differing levels of access to its resources. The administrator of "greatmachine" wishes to grant the same level of access to ten expected service requestors -- "zathras01" through "zathras10" -- who execute on other nodes within the cluster. The administrator can choose to set up an access control list that contains each one of the ten service requestors all with the same access privileges, or the administrator can choose to set up a group for these ten service requestors. Either choice is valid, and either choice will work. The administrator may chose to use the group approach if the list of service requestors with common access privilege will increase or grow very large, and the administrator wishes to make the addition of new requestors easier. To make use of group access control, the administrator does the following:

- Creates a local operating system identity on the "epsilon3" system for each of the ten service requestors (ex: "ctzath01", "ctzath02", etc.).
- Associates the local operating system identity for each requestor to a local operating system user group created explicitly for access control purposes (ex: "zathras")
- Map the local operating system identities for the service requestors to their security identities.
- Create one access control entry for the "greatmachine" resource, using the local operating system group name ("zathras") to grant access to these ten service requestors.

CtSec provides convenience scripts and commands for setting up these local operating system identities and groups, and for associating the operating system identities to the security identities.

1. Currently, only DCE provides a mechanism to group security identities together.

4.2.2.1 CAUZ Configuration Information

CtSec's CAUZ component is designed to permit the eventual support of Enterprise Identity Mapping, a facility proposed for OS/400 and currently under consideration by IBM as a solution for mapping individual identities to identity groups for all of IBM's operating system platforms. Since Enterprise Identity Mapping is not yet a reality for AIX or Linux software, CtSec chooses to use the local operating system identity and group mapping mentioned in the preceding section. However, CtSec is designed to permit the administrator to override this behavior by configuring CtSec to use Enterprise Identity Mapping. The setting of this override is the sole configuration task needed for CtSec CAUZ, and is only needed if Enterprise Identity Mapping is available.

The initial release of CtSec does not define the format of the override setting. This override will be specified by the version of CtSec that provides Enterprise Identity Mapping support.

The CAUZ component is provided as part of the CtSec software, and consists of code developed within the laboratory.

4.3 CtSec Configuration Procedures

CtSec requires knowledge about the system's configured security mechanisms and the location of libraries for these mechanisms in order to function properly. The list of Linux or AIX cluster component trusted services is also required. This information is provided to CtSec during CtSec configuration, a semi-manual set of procedures that occurs after the installation of CtSec and before the cluster officially starts.

CtSec requires this information to be locally accessible to each node in the cluster. Because the cluster cannot be guaranteed to provide a default distributed data repository to assist in the distribution and access of this information, the data must be recorded locally to all nodes in the cluster. Documented procedures and a set of semi-manual convenience scripts are provided to assist the system administrator in recording the configuration information on each node.

A possible representation of a cluster if given in the following diagram.

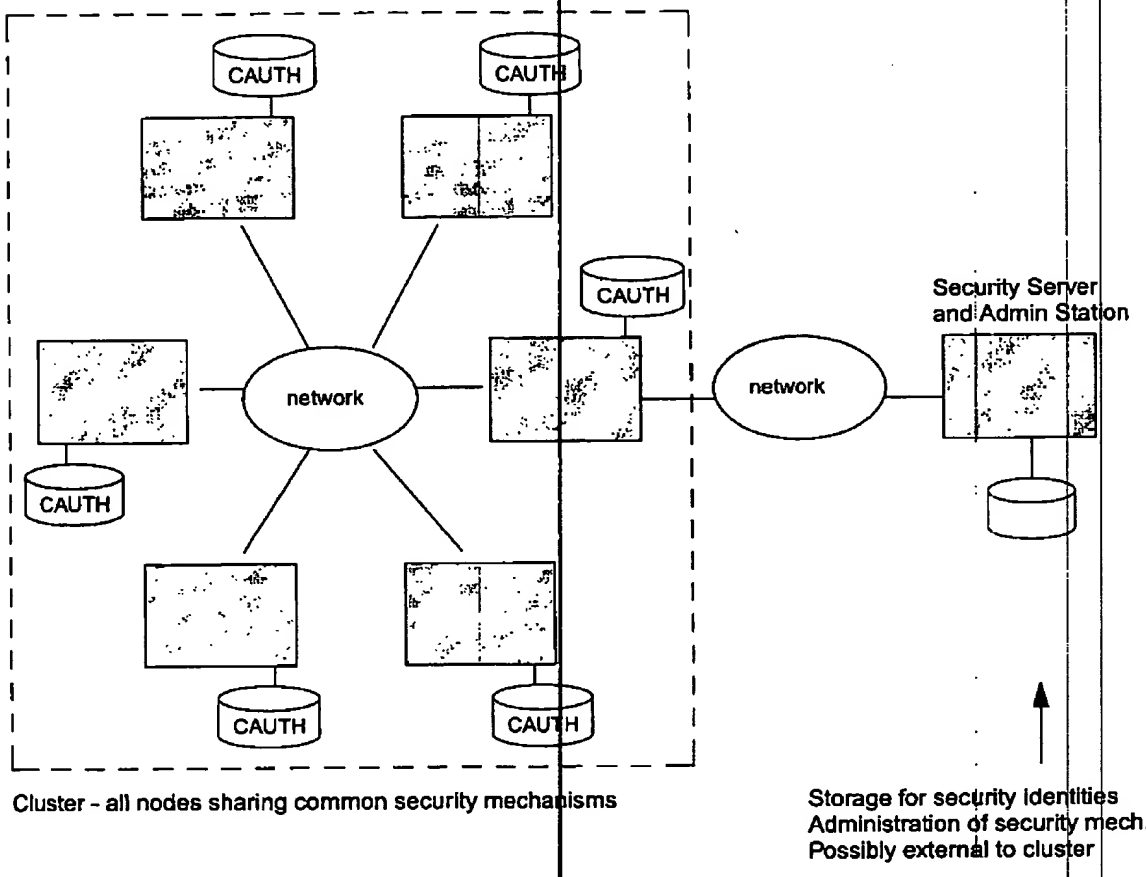

FIGURE 5 Example layout of a cluster and a network attached security administration workstation.

To configure CtSec, the following data must be identified:

- Identifying the security mechanisms configured on the system. Because all nodes within the cluster must use the same security mechanisms, this list will not vary from node to node within the cluster. The list of mechanisms must be consistent throughout the cluster.
- Identifying the mechanism pluggable modules (MPMs) associated with the security mechanisms. Since this list of security mechanisms is consistent throughout the cluster, this list of modules and libraries should also be consistent.

- Identifying the trusted services within the cluster. CtSec installs a file containing the list of trusted services shipped by IBM for clusters. This file may be modified by the system administrator to include customer written applications and trusted services from other sources.
- If group based authorization is to utilized by AIX or Linux Cluster trusted services, the creation of local operating system identities and groups for use in this authorization.

From this information, the CtSec configuration procedures and utilities perform the configuration tasks:

1. Creating entries in the configured node's CtSec CAUTH configuration file to indicate the security mechanisms available to the node and the corresponding MPMs provided for those mechanisms. This step involves the modification of files local to the configured node, so the procedure must be performed on the node being configured. This information should be the same for all nodes within the Linux cluster, since all nodes will have the same set of security mechanisms configured.
2. Creating security identities for the trusted services with the configured security mechanisms. For example, if Kerberos 5 is the underlying security mechanism, the trusted services will require Kerberos 5 principals to be created for them; if DCE were employed, the trusted services would require the creation of DCE UUIDs.

   For most security mechanisms, this task involves the creation of these identities within the security mechanism's network server. System administrators may wish to create these identities directly on the security mechanism server itself, and transfer these identities to the node afterwards. This server may not be part of the Linux cluster, but rather a network attached node. To create these identities directly on the node being configured, the node must have network connectivity to the security mechanism's network server and client software for that mechanism that permits creation of these identities from a remote system.
3. Creating security identity groups for use in group based authorization. Security identities are assigned as members of security identity groups, and this information is stored in a distributed repository. This is an optional step that can be performed only if group based authorization is desired.

   This task does not involve the creation of security identity groups within the security mechanism's network server, even if the security mechanism provides group support (such as DCE). Security identity grouping is achieved by associating security identities to local operating system identities on the nodes where service providers reside, and associating these local operating system identities with local operating system identity groups.

   Should the security identity groups already exist, the procedure uses the existing groups and add any new security identities to these groups.

4. Creation of service keys and keyfiles for trusted services within the cluster. This task requires that the security identities for the trusted services be previously created.

   This is another task that may be preformed directly on the node being configured, or on the security mechanism's network server. If done locally, the need to transmit the keyfile to the node being configured is removed. Otherwise, the keyfile must be securely transferred to the configured node.

5. Distribution of the keyfiles for trusted services to the correct nodes within the cluster. This task is performed only when a node's service keys and keyfiles have been created on another node in the network.

   Because keys are considered sensitive information, a secured means for transferring this information between nodes is required, and this method must not require the CtSec infrastructure to be operational at that time.

The first two tasks are performed on the node being configured. To lessen the chance of error injection during these two procedures, convenience scripts are provided to assist the administrator in creating the necessary entries, using the list of cluster components as input.

The remaining tasks in the above list can be performed anywhere within the cluster or on a remote security administration workstation, so long as the tasks are completed before the cluster components are made available for general use. Convenience scripts are also provided to assist in the creation of security identities and groups to make this task more reliable.

4.4 Functional Flow

There are five basic functional flows during the operation of the CtSec Infrastructure:

- CtSec client application initialization
- CtSec client establishes itself as a service provider
- Establishing a secured environment between service requestor and provider
- Transmitting secured information between service requestor and provider
- Verifying authority of a service requestor

Mention will be made periodically to *insecure* cluster configurations and to *weakly secured* cluster configurations. *Insecure* configurations do not make use of either operating system assurances nor "trusted" third party assurances that either the service requestor or the service provider are authentic. Users in such configurations are considered to be unauthenticated, and to provide any service at all, trusted services that authorize service requestors must be configured to grant some measure of access to unauthenticated users. *Weakly secured* configurations make use of operating system based assurances alone that the parties are authentic. Some assurances are reasonably reliable, provided the integrity of the operating system can be assured. Protections on Unix domain sockets, which assure that only authentic users on the local operating system are accessing the socket as either provider or requestor, can be considered reliable. Other methods, such as trusting users from known hosts, are considerably less reliable and open to host "spoofing" attacks, but are better than nothing if a "trusted" third party verification system cannot be installed or configured.

The choice of using the default *secured* configuration, *weakly secured*, or *insecure* configurations is left to the system administrator. However, use of any other configuration other than *secured* is *strongly* discouraged unless circumstances make it impossible or impractical to implement.

The discussion of these functional flows will use descriptions of possible failure scenarios. Terminology used to discuss these scenarios is taken from the IBM Unix Development Laboratory's "Software Serviceability Policy". The following is a short summary of this terminology:

External Failure	The source of the failure is external to the software. This includes incorrect usage of software interfaces and execution of the software in environments where the software is not designed to execute.
Internal Failure	The source of the failure is internal to the software. This include "bugs" within the source code, incomplete error checking, and treating of failure cases as successful cases.
Resource Failure	The source of the failure is associated with a resource required by the software that is not under the control of the software. Examples of such failures are memory allocation failures, input/output failures, missing files, and network connectivity problems.

4.4.1 CtSec Client Application Initialization

To make use of the CtSec Infrastructure functions, an application must initialize the CtSec Infrastructure for its use. Initialization involves the creation and setup of key data structures used by the infrastructure and the loading of the CtSec Mechanism Pluggable Modules (MPMs) that are installed. Loading these modules at this time causes the overhead of loading to occur during the initialization stage, instead of later during an authentication stage where timing may be important to the application. Because this data allocated by CtSec is for the infrastructure only, the CtSec client receives only a handle to the data structures, which is called a *security services token*. Within the token, mechanism independent information and state information is stored. This token is required by all other interfaces of the CtSec Infrastructure MAL.

Initialization is handled by the MAL Interface layer and the CAUTH layer. CAUTH is used to load the CtSec MPMs. Failure in initialization are not recoverable, meaning that the application will be unable to use CtSec until another initialization succeeds.

Weakly secured clusters will load an MPM that performs operating system authentications based on operating system user. If the cluster is being operated in an insecure mode, no MPMs will be loaded. In either case, a *security services token* will still be created so that trusted services can continue to use the remaining CtSec Infrastructure calls for transactions between service requestors and service providers.

Assuming that the development effort eliminated all sources of internal failures, the most likely sources of failures in this effort are:

External Failure	Incorrect usage of the MAL interfaces by the CtSec client application.
Resource Failure	Memory allocation failure, module loading failure, CAUTH configuration file missing or corrupted.

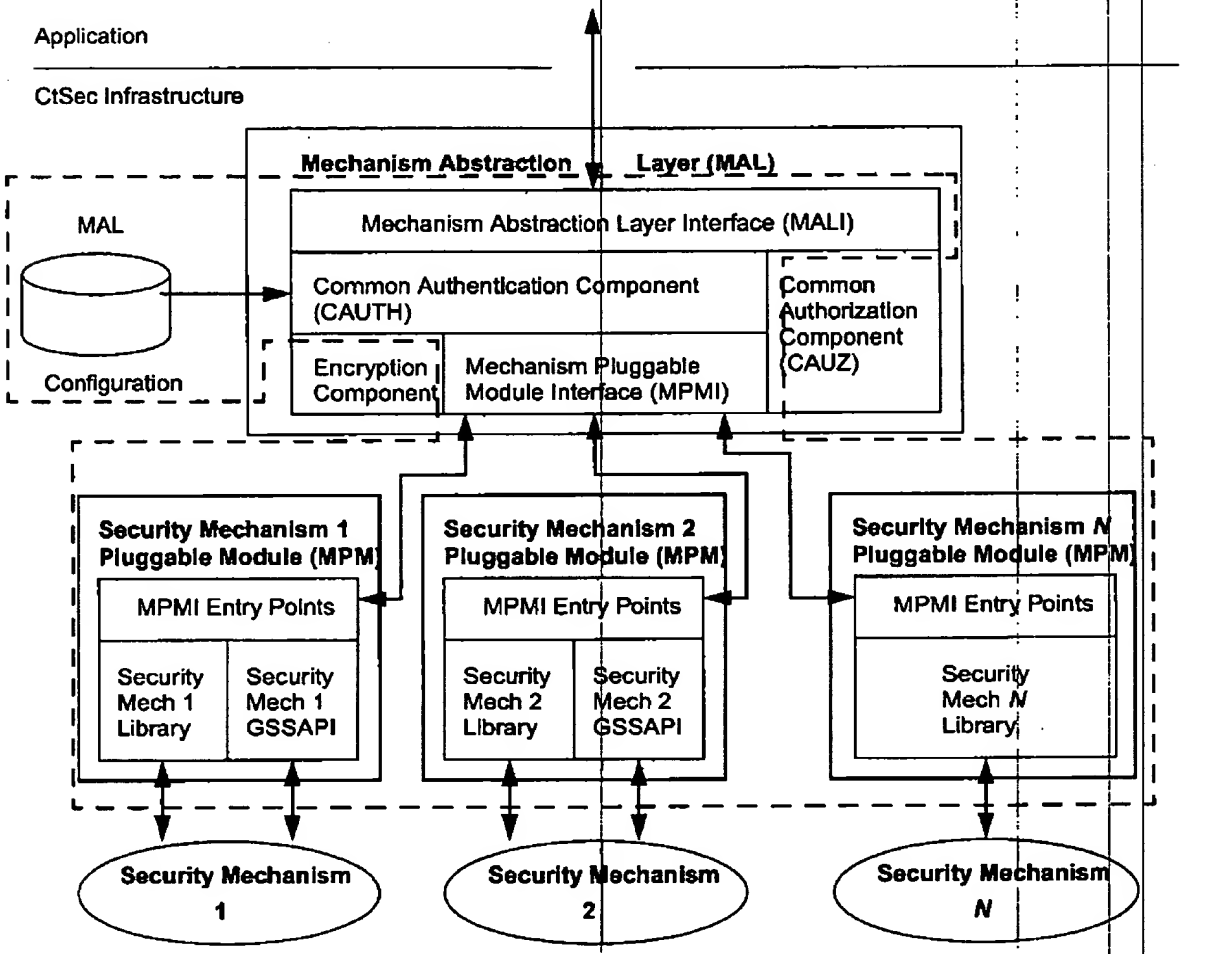

FIGURE 6 Components of the CtSec Infrastructure involved in CtSec Client Application Initialization

Applications designed for a networked client/server model should continue to "CtSec Application Establishes Itself As A Service Provider" on page 28. Local system client/server applications should continue to "Establishing a Secured Environment Between Local Applications" on page 38. Applications using distributed peers that wish to authenticate their peer components should continue to "Establishing a Secured Environment Between Application Peers" on page 36.

4.4.2 CtSec Application Establishes Itself As A Service Provider

Some service providers require that their clients authenticate themselves through the use of a third party that the service provider trusts. Even though the client itself may not be initially trusted, if the service provider can authenticate the client using the trusted third party, the client will be considered trustworthy by the service provider. This method is explained in "Requestor-Provider, Third Party Brokered Authentication" on page 47.

To authenticate with trusted services within a cluster through such a third party, or to mutually authenticate with applications or client users, a CtSec client application must obtain credentials. Some applications make use of the credentials inherited from their end-user. Other applications, such as trusted services which tend to be launched automatically by the system from `init` or `inetd`, have no end-user from which to inherit credentials. These applications must therefore acquire their own credentials.

Applications obtain a security identity and security credentials in order to utilize the underlying security mechanisms. This provides the application with sufficient privilege to perform simple and mutual authentication, and to make use of the underlying security mechanism facilities. Permission to use the security mechanism facilities may be required if the component design chooses to use a security mechanism feature for associating a security identity to a local operating system identity on a node for group authorization purposes. Credentials are stored in the process's private data structures that are used and maintained by the operating system.

CtSec CAUTH is used to obtain this permission to use the security mechanisms and establish the credentials. CAUTH instructs the MPMs for each security mechanism to obtain credentials for that specific security mechanism. This process is initiated by the CtSec client application, using the MALI interface for establishing a service provider.

In insecure and weakly secured clusters, no security mechanism is configured, so no permission is needed to use an underlying security mechanism. In effect, the CtSec Infrastructure converts this step in the process to a no-op. This permits a service provider to utilize the same code path for both secured and unsecured clusters. When operating in insecure clusters, all service requestors will appear to the service provider as unauthenticated users. Any service provider attempting to authorize a service requestor must be capable of granting access to unauthenticated users.

Internal sources of failure should be eliminated during the implementation and testing efforts. Possible sources of failure at this level include:

External Failure	Identification failure (incorrect service name or service key provided), interface usage failure
Resource Failure	CtSec CAUTH configuration file missing or corrupted, MPM module loading or location error, security context memory allocation failure, security context creation failure (data not accessible from MAL)

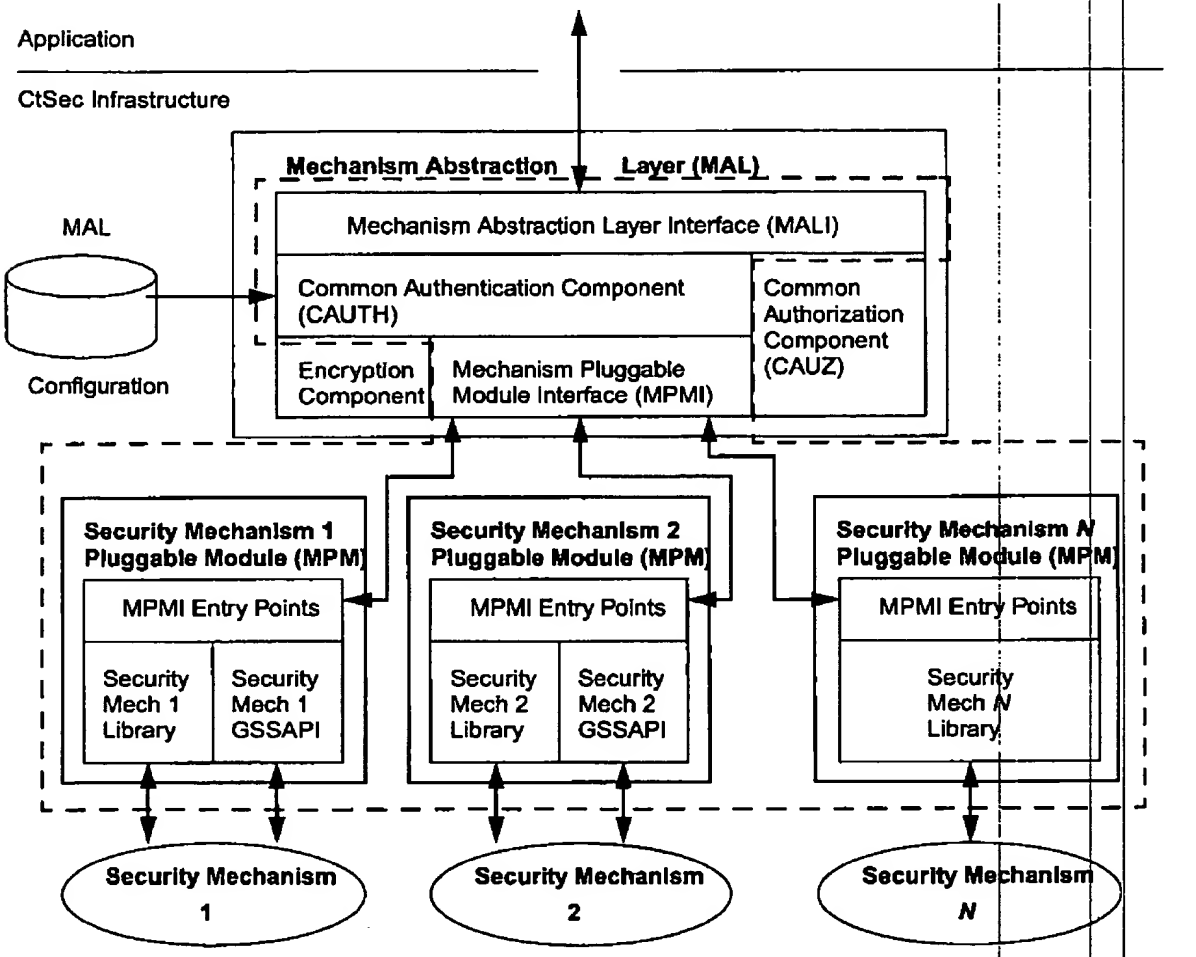

FIGURE 7 Components of the CtSec Infrastructure involved in trusted service establishment

4.4.3 CtSec Application Establishes Itself As A Service Requestor

Some service providers require that their clients authenticate themselves through the use of a third party that the service provider trusts. Even though the client itself may not be initially trusted, if the service provider can authenticate the client using the trusted third party, the client will be considered trustworthy by the service provider. This method is explained in "Requestor-Provider, Third Party Brokered Authentication" on page 47.

In order to make use of the trusted third party, the service requestor must first verify to the trusted third party -- the security mechanism -- that it is authentic. This is done by "logging in" to the underlying security mechanism, by identifying itself as a party that the security mechanism already knows.

CtSec CAUTH is used by service requestors to identify themselves to the underlying security mechanisms. The application invokes the CtSec MALI routine provided for service requestors to establish their security identity. When successfully completed, the application will have established both its identity and credentials for using the security mechanism to authenticate itself with a service provider. CtSec CAUTH invokes the MPMs for all available security mechanisms, instructing these modules to obtain the security identity and credentials for this process for that specific security mechanism. This information is stored in the memory location referenced by the *security services token* that was established previously.

In both insecure and weakly secured clusters, the CtSec Infrastructure will not obtain a security identity for the application. This permits the service requestor to use the same code path for both secured and unsecured clusters. The service requestor will possess the same privileges as an unauthenticated user in an insecure configuration. Weakly secured configurations will make use of the service requestor's operating system user identifier for authentication purposes.

Internal sources of failure should be eliminated during the implementation and testing efforts. Possible sources of failure at this level include:

External Failure	Identification failure (incorrect service name or service key provided), interface usage failure
Resource Failure	CtSec CAUTH configuration file missing or corrupted, MPM module failure, security context modification failure

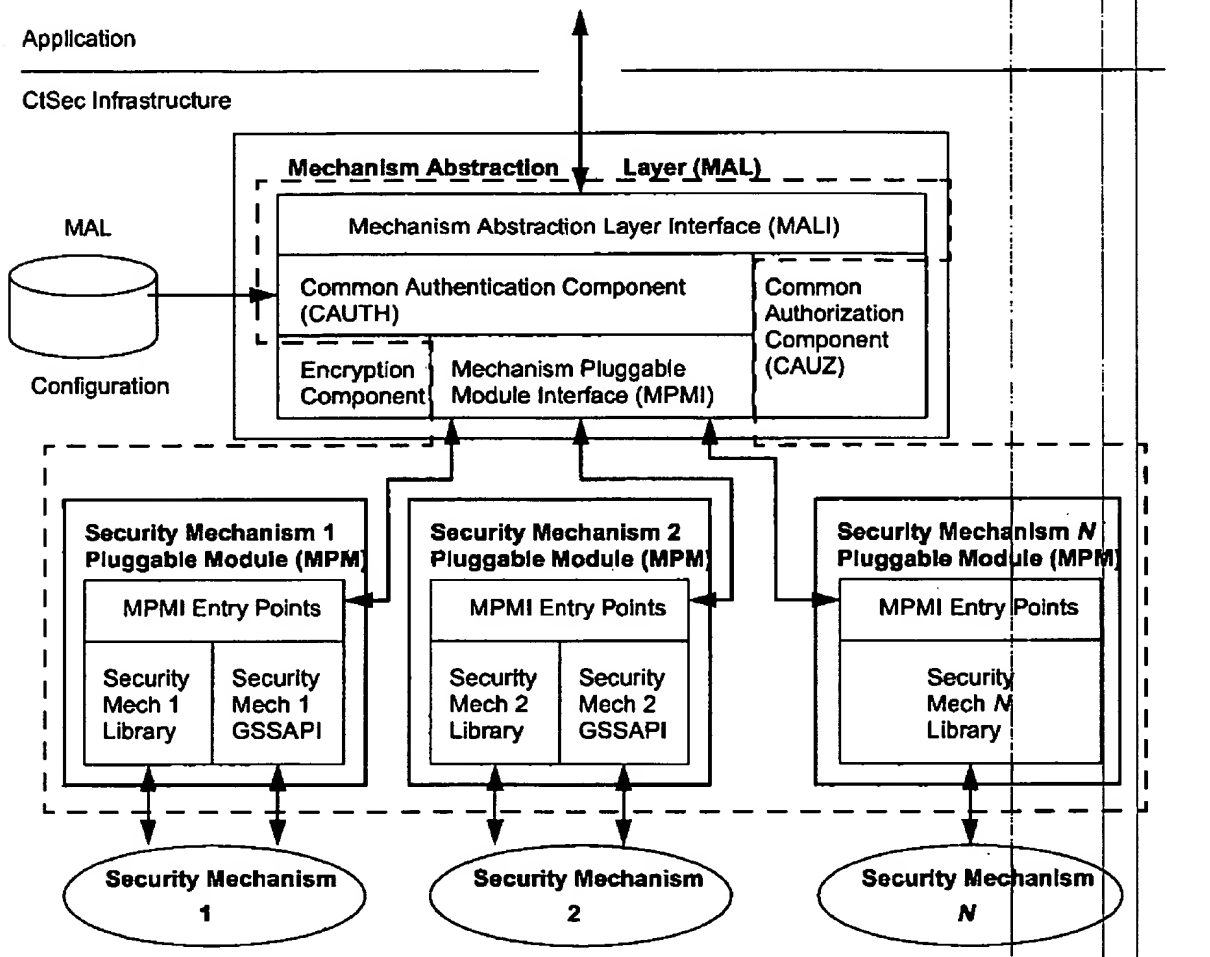

FIGURE 8 Components of the CtSec Infrastructure involved with service requestor establishment.

4.4.4 Establishing a Secured Environment Between Service Requestor and Provider

Once an application has obtained credentials, either by using the credentials established by its end-user or by obtaining its own credentials via CtSec, the application can attempt to authenticate itself with another application or trusted service. Once this authentication succeeds, a secured environment is established between the application -- as service requestor -- and the service provider. This environment is named the *security context*. Once the security context is established, the service requestor and service provider remain authenticated for the duration of the context. Should the context expire or the

connection established between provider and requestor be dropped, a new authentication is required to establish a new security context.

Authentication is a "hand shaking" process which requires the requestor and the provider to agree on the security mechanism they will use, and to agree that both parties are trusted by the security mechanism.

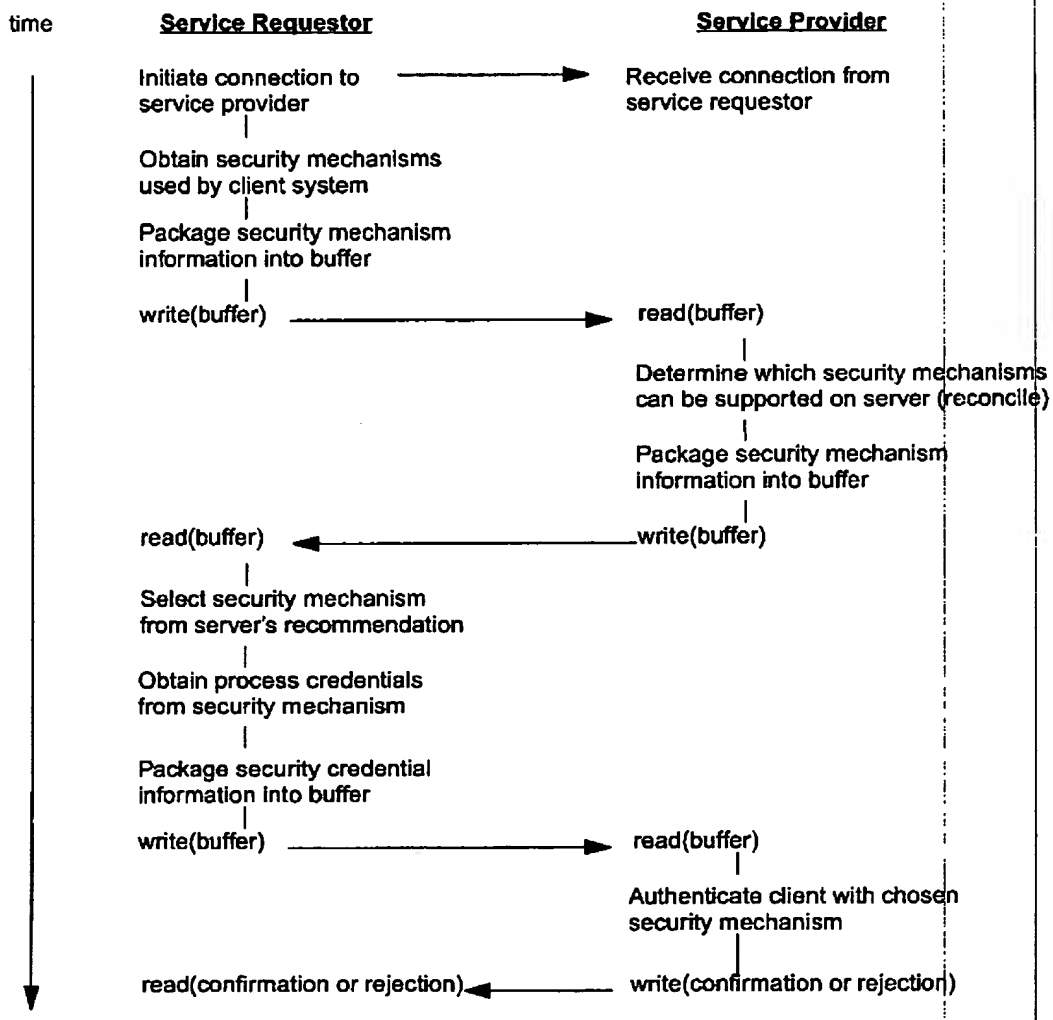

FIGURE 9 Requestor-Provider Authentication Flow, Single Authentication Model

CtSec's CAUTH component provides the utilities for both service requestors and service providers to perform these authentication tasks, for both single and mutual models of authentication. Given CtSec's overall goal of abstracting the security mechanisms used by the system from the service requestor and provider alike, CAUTH determines what security mechanisms are available, which ones to select both during provider reconciliation, and ultimately which mechanism will be agreed upon as the one to be used for these parties.

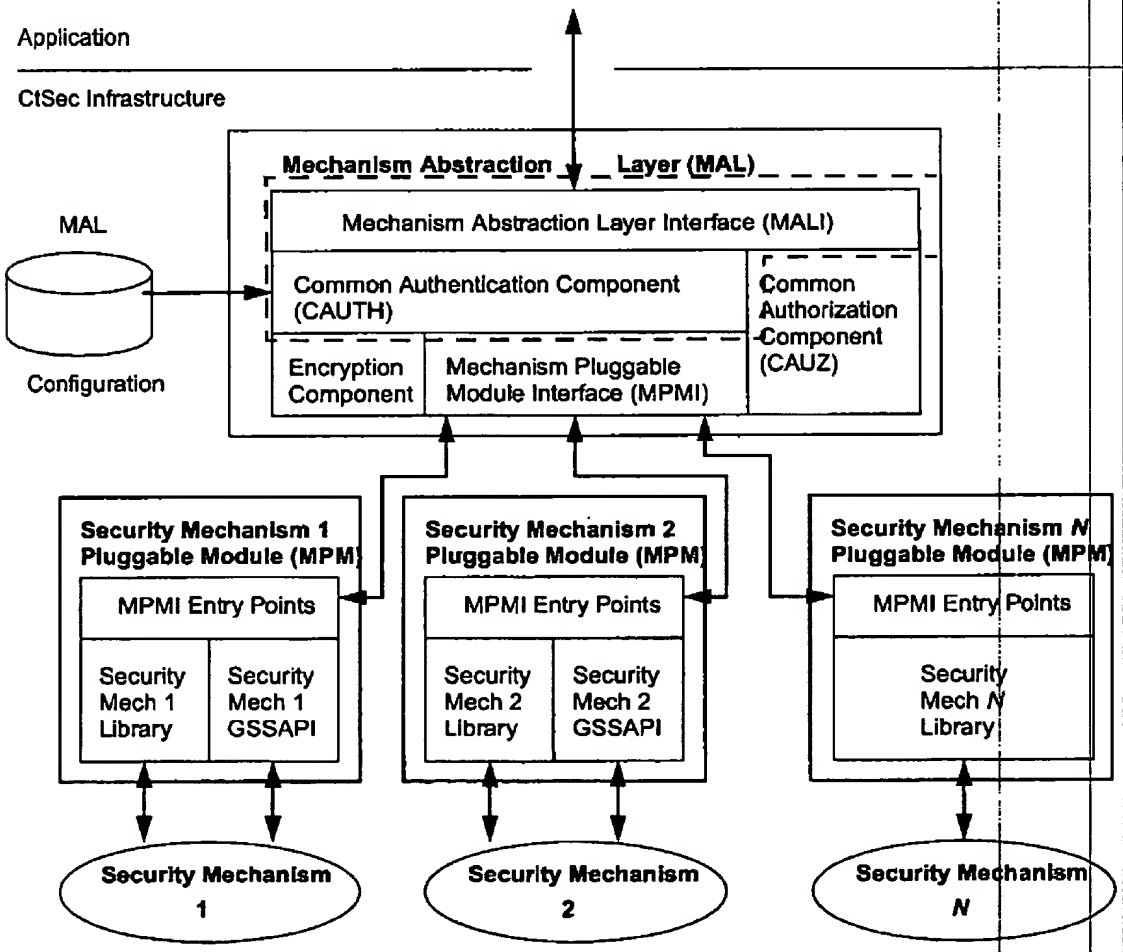

FIGURE 10 Components of the CtSec Infrastructure involved in establishing a security context.

For the service requestor, CAUTH abstracts the instructions required to obtain the list of security mechanisms supported by the client's system. This information is returned in an opaque data struc-

ture, which the requestor can then transmit to the service provider. This data does not require encryption, since the information itself is not protected.

The service provider reads the opaque data structure, and in turn uses CAUTH routines to perform the security mechanism reconciliation; CAUTH examines the list of security mechanisms, compares it to the security mechanisms supported by the server's own system, and returns an opaque data structure that contains the list of mechanisms common to both systems. The service provider is not involved in the reconciliation process. CAUTH obtains the information it requires from the CAUTH configuration information. The provider then transmits the opaque data structure to the service requestor. Since this information is not protected, it can be transmitted without being encrypted.

Upon receipt of the reconciled security mechanisms, the requestor uses CAUTH routines to select one of these mechanisms and to obtain the requestor's credentials associated with that mechanism. The requestor makes no explicit selection of what mechanism to use; CAUTH makes this decision. Credentials are obtained using GSSAPI routines associated with the security mechanism chosen by CAUTH, packages in an opaque data structure, encrypts the buffer in a manner that can be decrypted by the service provider, and provides this buffer to the requestor. The requestor must then transmit the opaque data structure to the service provider to authenticate the requestor's identity.

At this point, a *security context* exists for the requestor, but not for the provider yet. A handle is provided to the requestor, called the *security context token*, which must be provided to any CtSec MAL calls the requestor uses to communicate with the service provider. The context contains information on the mechanism being used for this security context, including keys to be used for encrypting and decrypting data transmissions in the future.

After receiving and decoding the requestor's credentials buffer, the service provider invokes CAUTH routines to extract the credentials from this buffer and authenticate the requestor using the mechanism selected by the requestor.

In single authentication models, a *security context* is established for the provider at this point, provided that the security mechanism successfully authenticates the requestor's identity. This completes the setup of the security context between the requestor and the provider. A handle is given to the service provider, called the *security context token*, which must be used by the provider in an CtSec MAL call to communicate with the requestor. The context contains security mechanism specific information. After this point, the requestor and the provider may commence their transactions in a single authentication model. However, these parties will not be able to encrypt and decrypt data between them unless they share a common secret key, or unless both parties decide to mutually authenticate with each other.

In mutual authentication models, the service provider must repeat the steps performed by the service requestor: obtaining credentials for itself and transmitting these credentials to the service requestor so that the requestor can validate the provider's identity. As with the requestor's case, CAUTH automat-

ically obtains these credentials from the correct mechanism, packages them in an opaque structure, and returns the structure to the provider. As is also the case with the requestor, the provider establishes a *security context* at this time, and the provider will use the handle to this context in future CtSec MAL calls during its transactions with the requestor. Unlike the single authentication model, the setup of the security context is not complete. The provider must transmit the opaque data structure back to the requestor, so that the requestor may authenticate the service provider.

After receiving the provider's credentials buffer, the service requestor invokes CAUTH routines to extract the credentials from this buffer and authenticate the provider. A *security context* is established for the requestor at this point, so long as the security mechanism successfully authenticates the provider's identity. This new context must be used *in place of* the context formerly established for the service requestor; the old context is to be discarded by the requestor. This completes the setup of the security context between the requestor and the provider in a mutual authentication model. A handle is given to the service requestor, called the *security context token*, which must be used by the requestor in an CtSec MAL call to communicate with the provider. Both the provider's and the requestor's security contexts will contain the keys necessary to transmit and receive encrypted information.

To send encrypted data between a service provider and a service requestor, both the client and requestor must mutually authenticate.

In weakly secured clusters, both the service requestor and the service provider agree to use only the operating system's assurances instead of a "trusted" third party to authenticate the identity of the other party. In practical terms, both the requestor's and the provider's systems are configured to use only the operating system to assure authenticity. The CtSec Infrastructure makes use of the provider's and requestor's operating system identity instead of credentials. When both the provider and the requestor are guaranteed by the operating system to reside on the same node, the flow described in "Establishing a Secured Environment Between Local Applications" on page 38 is used to obtain this assurance. When this assurance cannot be provided, the remote operating system is contacted and requested to verify the identity of the remote party. This assurance assumes that both the network and the remote operating system can be trusted, and such assurances are minimal unless the network and the nodes are physically isolated.

Simple and mutual authentications are a formality in insecure clusters. A *security context token* is always established, but no session key is established for the context. Requests to encrypt data for secured transmission result in a "clear" message being returned to the caller, and decoding these messages on the other side of the connection results in merely extracting the "clear" message. Service requestors and service providers are oblivious to this, since they continue to use the same CtSec Infrastructure routines for secure data transfer in both secured and unsecured clusters.

Failure scenarios in this model include:

External Failures (Requestor)	Usage failures in CtSec MAL calls
External Failures (Provider)	Authentication failure of requestor on provider's system

Resource Failures (Requestor)	Inability to obtain credentials from underlying security mechanism, no common security mechanisms between provider and requestor, authentication failure of requestor on provider's system, mutual authentication failure of provider on requestor's system, memory allocation failure on requestor, memory allocation failure on provider
Resource Failures (Provider)	Inability to obtain credentials from underlying security mechanism in mutual authentication cases, authentication of provider on requestor's system in mutual authentication models, memory allocation failure on provider, memory allocation failure on requestor in mutual authentication models

4.4.5 Establishing a Secured Environment Between Application Peers

Not all authentication requires the use of a trusted third party. Some distributed components of the same application make use of a common, shared secret key. Knowledge of this key is sufficient proof to one component of the application that the other component is trustworthy. This method removes the need to use a trusted third party, so these components would not need to obtain security identities and credentials for use between their own components. However, such applications still require security mechanisms to encrypt and decrypt information using keys. This method is explained in more detail in "Daemon-to-Daemon, Common Secret Key Verification Authentication" on page 48.

In this method, each component within the application has access to the same common, shared secret key. CtSec provides a means for the application to obtain its correct secret key. CtSec also ensures that the keys are kept current for the application, and that the application always has access to the most current and the next most recent keys. The CtSec MAL will determine if new keys need to be obtained and which key needs to be used for deciphering incoming data as a natural course of the interfaces for sending and deciphering data. The application does not need to perform its own key management; this task is assumed by the CtSec MAL. The component design will demonstrate how this is accomplished by the CtSec MAL.

CtSec will permit these components to encode and sign messages using the key, and for the other components to decode and verify these messages. Applications invoke the CtSec MALI routine to either encode or decode messages, and the CtSec MAL responds by invoking encryption routines available in the operating system to perform these functions. DES and Triple-DES are examples of such encryption software.

Establishing a secured environment through this method does not depend upon the presence of a trusted third party such as Kerberos version 5. Therefore, this method can be employed even in weakly secured and insecure clusters, provided that some sort of encryption software such as DES or Triple-DES is installed on all nodes in the cluster. If no encryption software is available, the CtSec

Infrastructure simply returns the "clear" form of the original message. Decryption of such messages also becomes a no-op, and verification of data signatures are always successful.

Internal sources of failure should be eliminated during the implementation and testing efforts. Possible sources of failure at this level include:

External Failure	MAL interface usage failure

Note that CtSec will be unable to detect whether the "correct" key has been used to either encode or decode the information. If the application uses the wrong key with the CtSec MAL routines, CtSec will be unable to detect this. Verification of the data is the responsibility of the application.

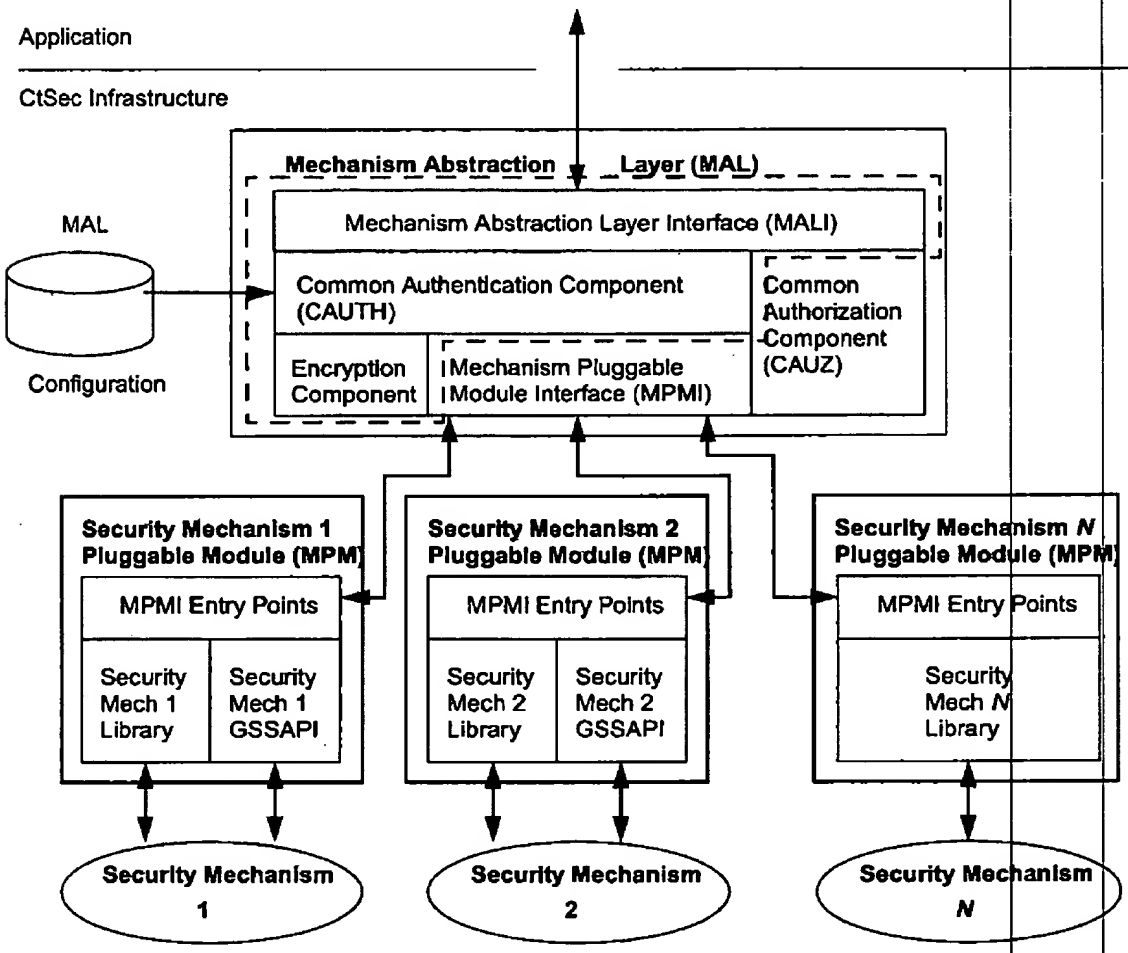

FIGURE 11 Components of CtSec involved in common secret key based authentication

4.4.6 Establishing a Secured Environment Between Local Applications

Another authentication model that does not require the use of a trusted third party can be used between a service requestor and a service provider when both parties are guaranteed to be executing on the same system. In these cases, the operating system's assurance that both parties are valid users is sufficient to trust the other party. Encryption and decryption of information can also be avoided if both parties use a private communication mechanism that cannot be simultaneously used by another process, such as a Unix domain socket connection between requestor and provider. This method id explained in "Operating System Based Authentication" on page 48.

Because this model does not require the use of the underlying security mechanism nor a DES encryption library, the CtSec component is not required to provide special function for applications using this method. For application convenience, CtSec does provide MALI routines to set up and access the Unix domain sockets that are primarily used in these scenarios.

Because this method relies on the operating system to prevent misuse of the Unix domain socket, this method of establishing a secured execution environment is always available from the CtSec Infrastructure, even in weakly secured and insecure cluster configurations.

Failure scenarios in this model include:

External Failure	MALI usage failure, insufficient permission to access Unix domain socket

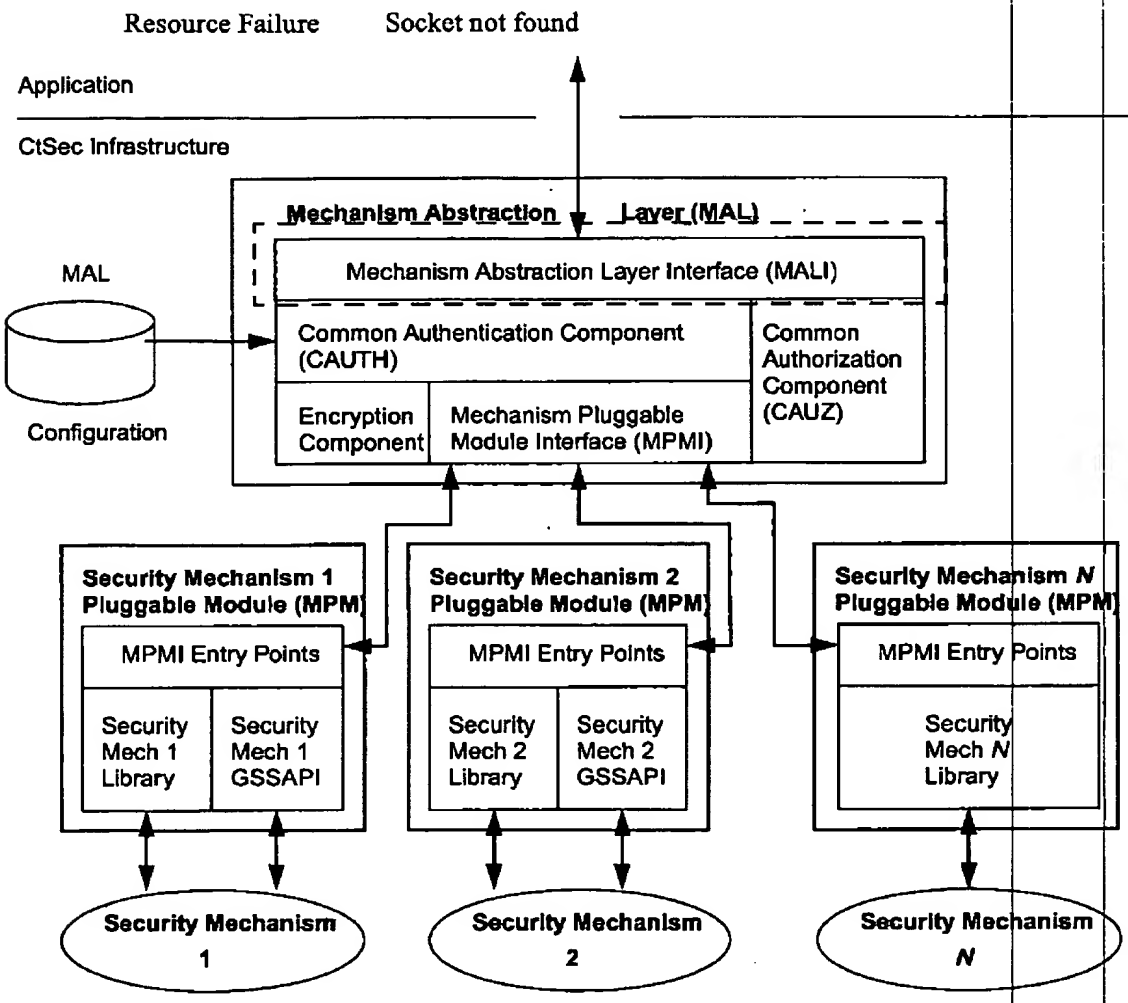

FIGURE 12 Components of CtSec involved in operating system based authentication

4.4.7 Transmitting Secured Information Between Service Requestor and Provider

After authentication has been performed and security contexts have been established[1] for the service requestor and the service provider, both parties are considered to be authentic system "users" and may begin transacting business between them. These parties may need to transmit information between

1. These contexts are established using one of the mechanisms described in the three preceding sections.

them that must be protected from use or examination by other parties which may be eavesdropping on the connection or intercepting the traffic on the connection. To prevent use of sensitive information from such nefarious external parties, the requestor and the provider can choose to encrypt the information in a manner that only they can decrypt.

CtSec provides two methods for processing data for secured transmission:

- *Security Context Based Encryption.* This method uses keys stored in the security context for both the service requestor and service provider to encrypt and decrypt the data. The CtSec MAL uses information stored in the security context data structures to encrypt and decrypt the information. Providers and requestors simply call the CtSec MAL routine to either encrypt or decrypt the information, providing the security context token as one of the argument to the call.
- *Private Key Based Encryption.*This method uses a common, secret shared key that both the requestor and the provider share. How these two parties came to share this common key is not the concern of CtSec; both parties used Some Other Means to obtain this key. The encrypting party provides the data stream and the common, secret shared key to use in encoding the data stream. The decrypting party provides the encrypted data stream and the same common, secret shared key.

Because the methods differ, two separate sets of interfaces are required for encryption. CtSec provides one pair of interfaces for security context based encryption, and a different set of interfaces for private key based encryption. Both sets of encryption routines use any available library routines for encryption and decryption of the data. Examples of such software include DES and Triple-DES.

Private key based encryption is available in secured and unsecured clusters, provided that encryption software such as DES or Triple-DES is made available on all cluster nodes for the CtSec Infrastructure's use.

Security context based encryption is not available in weakly secured and unsecured clusters, although this fact is hidden from the CtSec Infrastructure client. Service requestors and service providers use the same CtSec Infrastructure routines to prepare and process data in either case. The CtSec Infrastructure will determine that the cluster is insecure, and forgo the encryption and decryption of the message.

The CtSec Infrastructure only encrypts and decrypts the data; it does not transmit this information to the other party. The CtSec client is still responsible for the transmission of this information.

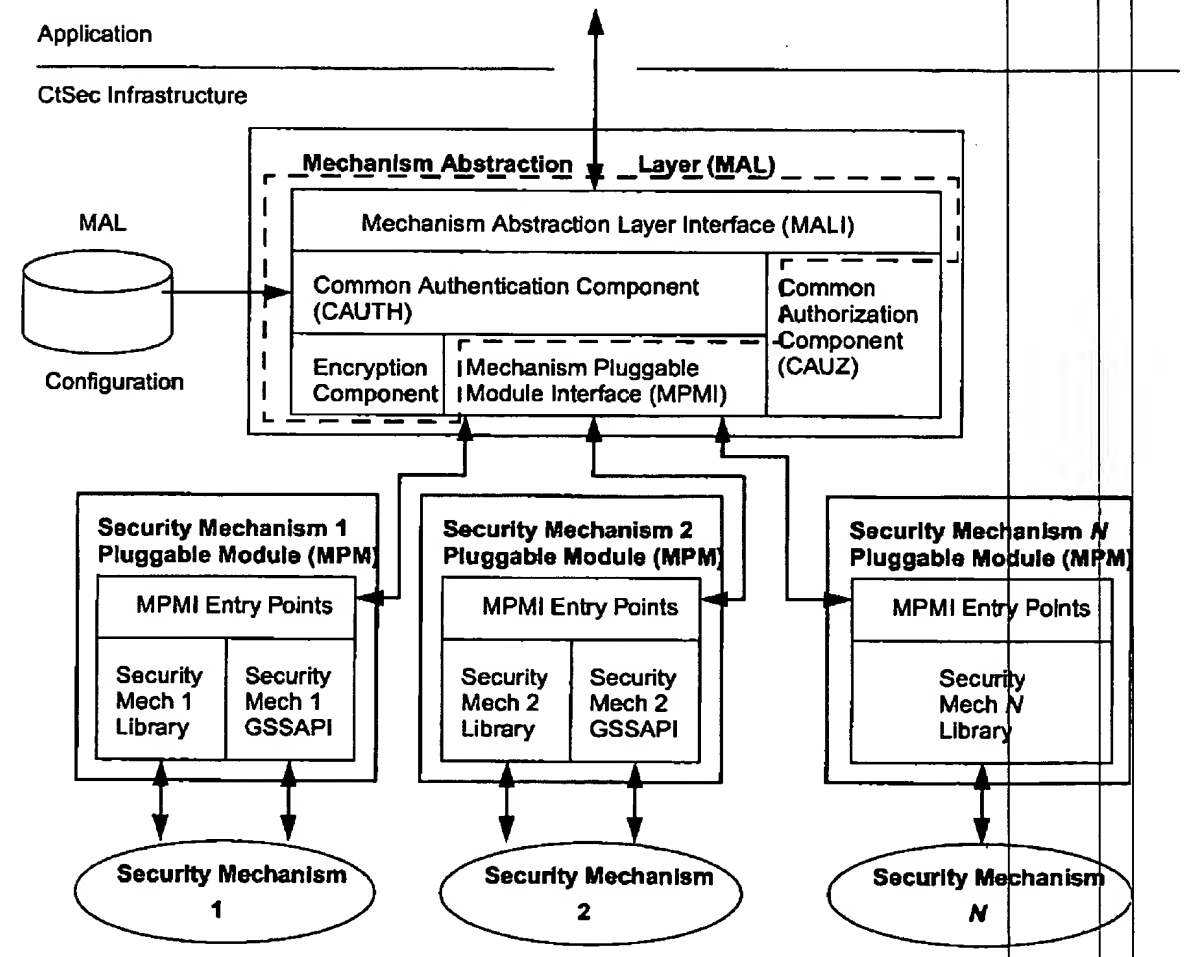

FIGURE 13 Components of the CtSec Infrastructure involved in encryption and decryption of secured information.

Failure detection during the encrypting or decrypting procedure is rather limited. The CtSec MAL is incapable of determining if the "correct" encryption key is provided by its caller, or if the other party in the security context is making use of the correct keys in either encryption or decryption of the information. Therefore, final assessment on how "correctly" the encryption or decryption of the information was performed is left to the service provider and requestor to determine. Failure scenarios that can be detected are:

External Failure Incorrect usage of CtSec MAL interfaces

Resource Failure	Memory allocation failure, DES encryption or decryption failure

4.4.8 Verifying Authority of a Service Requestor

On occasion, service providers may wish to restrict certain function or resources from most users, granting full or partial access to a subset of valid system users and trusted services. In addition to being an authentic system user or trusted service, the requestor must also meet specific access criteria, which are granted based on the user or trusted service's identity. In other cases, resources or functions may instead be restricted from selected users or trusted services, while access is granted to all other authentic users and trusted services. Service providers use access control to protect such resources and functions. Access control is traditionally granted using two basic schemes:

- Granting or denying access based on the requestor's identity
- Granting or denying access based on the requestor's group association.

The first method is rather simple, straightforward, and supported by most security mechanisms: the service provider maintains a list of those users and services who are granted (or denied) access to the resource or function that the server provides. While simple, this strategy becomes cumbersome to administer in large systems with many users or trusted services and many resources or functions to control. Performance is also impacted in such environments, where access control checking becomes a factor of system's size and the user community's size. Also complicating this approach is the tendency for different security mechanisms to provide differing solutions to individual access control.

The second method is supported by some security mechanisms: users and trusted services are assigned to membership within specific user *groups*, and group membership becomes part of the process's identity much as the user's identity. Instead of maintaining a list of all possible users, the service provider maintains a list of the groups supported by the system to which it will grant access. Instead of looking for the user's or service's identity before granting access, the user's or service's group membership is examined, and access is granted or denied based on that membership. Access control checking is more scalable in this approach, but not all security mechanisms provide group membership assignment of their security identities.

4.4.8.1 Identity Based Authorization

To grant access based on identity, the service provider must decide what resources or functions it controls, determine which ones require special access, and create an *access control list* (ACL) to contain this list.

4.4.8.2 Group Based Authorization

Because security identity grouping is not a feature supplied by all security mechanisms, CtSec provides its own scheme for providing group access through the use of Unix operating system based user identity groups. In cases where the underlying security mechanism does provide a grouping scheme, CtSec will not exploit it. This allows CtSec to avoid possible assumptions that are mechanism specific from surfacing in the mechanism abstract group mechanism provided by CtSec.

4.4.8.3 Authority Verification in Weakly Secured Clusters

The requestor's operating system identity is used to verify the requestor's access in weakly secured clusters. In these clusters, the access controls maintained by the service providers must make use of operating system identifiers.

4.4.8.4 Authority Verification in Insecure Clusters

In insecure clusters, all service requestors will appear to the CtSec Infrastructure client as unauthenticated users. All unauthenticated users are handled the same way in authorization verification, and granted whatever level has been granted to the general classification of unauthenticated users.

System administrators need to understand the nuances in this case. In effect, insecure clusters remove the service provider's capability to grant differing levels of access to different users. Service provider applications that use access controls must allow access to unauthenticated users, and basically grant this class of user full access to the service. If no entry exists in the access control list for the unauthenticated user class, all users will be denied access to the service in an insecure cluster.

System administrators must also realize that the access controls need to be revised if the cluster is migrated from insecure to secure. The full level of access should no longer be granted to all unauthenticated users in this case. Such a migration will necessitate shutting down the trusted services, modifying the access controls to remove full access from unauthenticated users, and restarting the trusted services.

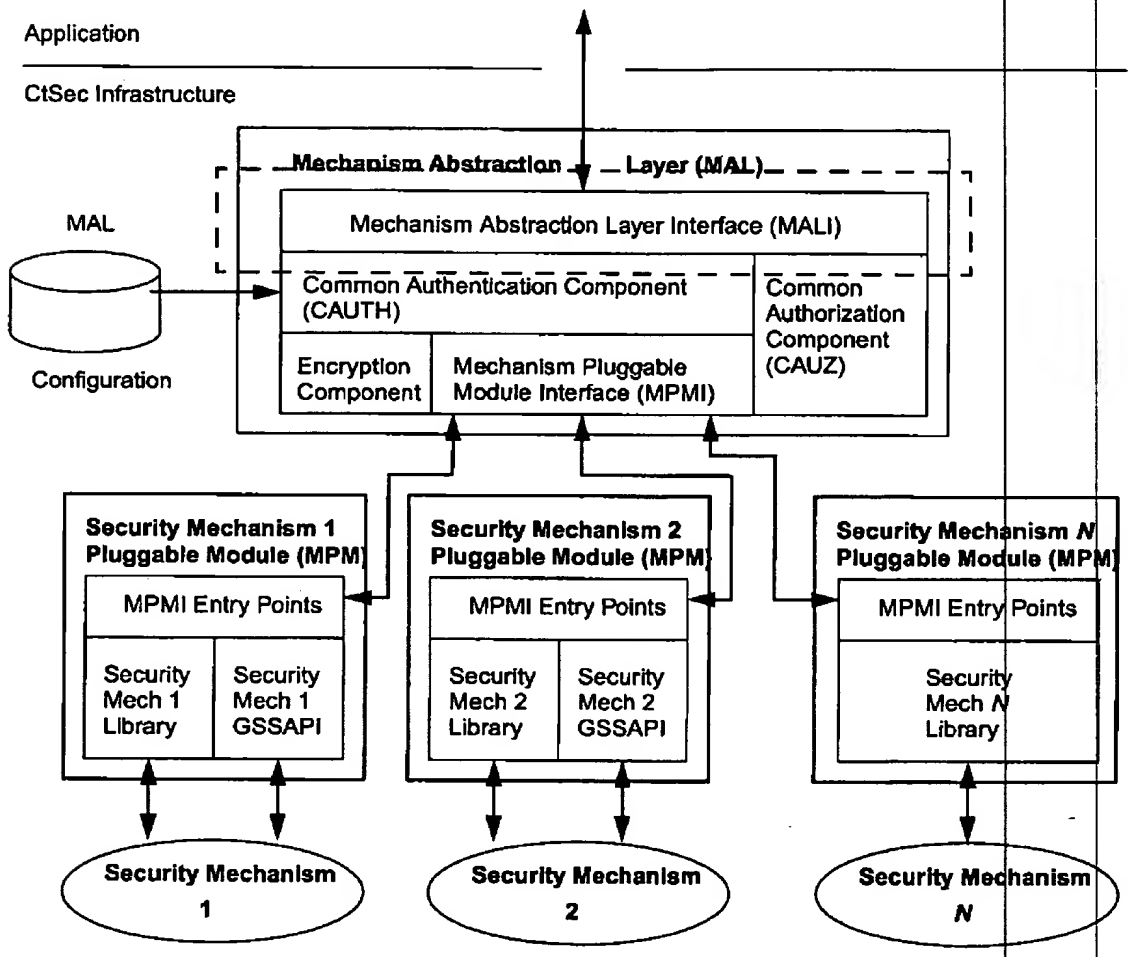

FIGURE 14 Components of the CtSec Infrastructure involved in authorization processing

5.1 Considerations in Secured Executions Environments

Implementing a secured execution environment involves *authentication* of service requestors and enforcing *access control* on resources controlled by the trusted services to prevent access by unauthorized service requestors.

5.1.1 Authentication

Authentication is the task performed by service providers to ensure that the service requestor is a recognized system service or system user, instead of a rogue party seeking access to system resources and function.

Trusted services use several programming models for authenticating service providers and service requestors. These methods tend to be based on common communication protocols used between the parties, although any model could be used in any communication protocol. The more common methods used in Linux cluster and SP software are described below.

5.1.1.1 Requestor-Provider, Third Party Brokered Authentication

Requestor-provider authentication requires the service requestor to authenticate itself with the service provider before requesting functions or resources controlled by the provider. Because neither party in this exchange is completely trusting of the other, a third party trusted by both is used to broker this authentication[1]. The model also allows for mutual authentication, where the service requestor can demand that the service provider also authenticate itself before the requestor accepts resources or function. To perform authentication in this model, trusted services and their client applications must consider the following:

- Include logic to start the CtSec security services, use CtSec to obtain the trusted service's security identity and credentials, and perform CtSec termination and cleanup logic prior to termination of the trusted service.
- Include flows in their communication protocols to accommodate the passing of security context data between the service requestor and the trusted service.

If a trusted service is unable to authenticate the service requestor, the connection to the service requestor should not be dropped. Instead, the service requestor should be considered an *unauthenticated* client, and the trusted service should continue to process the request. It is valid for some trusted services to grant some degree of limited access to unauthenticated users, and some access controls will permit access to resources and function for such users.

Security mechanisms are essential to this method of authentication. It is the security services that provide the identities used by the parties in this authentication method, and provide the means by which the identities can be verified. Users of this authentication method are therefore requires to use the

1. Examples of such third parties are Kerberos version 5 and DCE.

CtSec services for obtaining their security identities and credentials, and to verify the credentials of the other parties in the negotiation.

5.1.1.2 Daemon-to-Daemon, Common Secret Key Verification Authentication

The preceding paragraphs assume a traditional client and server execution environment utilizing a third party to broker authentications. This is necessary because the client and server are usually different applications, and neither can be assured of the identity of the other party without confirmation from a "trusted" third party. Not all secured execution environments follow this model.

IBM Linux and AIX clusters provide distributed subsystems that act as trusted services. These services are implemented as daemons that execute on all nodes within the cluster. In essence, the same application executes on each node. To provide their service to their clients, these distributed daemons often need to work together, with a daemon on one node making requests of its counterparts on other nodes in order to complete this function. These daemons still need to verify the identity of the other party to ensure that another application is not attempting to misuse the service, but a "trusted" third party is not necessarily required in this case. So long as all the distributed daemons can agree on a private means for identifying themselves, and in a manner that cannot be imitated by outside applications. This model is typically labelled *daemon-to-daemon* authentication by the cluster trusted services.

Typical implementations of this model utilize a common secret key, of which all instances of the daemon are aware. This common secret key is used to generate a message digest that is sent along with the message to the other instances of the trusted service. If the digest is successfully decoded and verified by the receiving party using the same common secret key, the transmitting party is considered to be authenticated by virtue of its knowledge of the secret club handshake.

Security mechanisms are involved in this authentication method to the extent that they provide the common shared secret key used by the daemons. Applications using this method use the CtSec interfaces to acquire the common shared secret key and rely on CtSec key management to keep the keys from expiring.

A problem inherent in this authentication method is the special processing that must occur when the common shared secret key is updated. This key has to be updated before it expires. Other security designs will deal with the management and maintenance of these keys.

5.1.1.3 Operating System Based Authentication

This model employs the operating system's assurance that both parties attempting to enter in a secured execution environment are authentic. No external verification of either party from the security service is requested.

One such authentication scheme present in all security configurations[2] uses Unix domain sockets to assure that both parties are authentic. In this model, a service provider establishes a Unix domain socket (UDS), which is only accessible to processes executing on the node; it is not possible for an

external application to access the socket. If the owner of the process possesses sufficient privilege to access the node, the service provider considers the requestor authentic. To help keep out casual users and other riff-raff, privileges are usually set quite high on these sockets so that only a very privileged set of users --such as *root* only -- can access the socket. Although this method of protection is a bit weak, it does improve performance by removing the need for an additional security service to be involved in the transaction between requestor and provider. The weakness is in trusting the operating system's integrity, but this concern is offset with the knowledge that if the *root* user is compromised, then there's a far greater security risk on the system than accessing a trusted service

Another authentication scheme makes use of operating system network services to verify the identity of parties connecting to a socket from a remote host. In this model, a service provider obtains the operating system identity and the host address source of the connecting requestor. The host address is verified and checked against a predefined list of trusted network hosts. If that test is satisfied, an identity providing service -- such as *identd* -- is contacted on the remote host and asked to verify the authenticity of the connecting party's operating system identity. At the successful completion of both tests, the connecting party is considered to be authentic. This protection is only as good as the level of trust that can be placed in the network and in every system operating on that network. If the network and its systems are not physically isolated, this scheme is susceptible to host address "spoofing" attacks.

5.1.2 Access Control

The implementation of access control may differ between the trusted services, because in the initial Linux cluster software release, CtSec will not provide access control list (ACL) storage facilities. Any trusted service that implements access control is responsible for developing a mechanism to store the ACLs it uses and load these ACLs during their startup procedures.

5.1.2.1 Common Characteristics of Access Control

All access controls have common characteristics which all trusted services can expect.

5.1.2.1.1 Association With Specific Functions or Resources

An access control list is associated with a specific resource or function. Any time a service requestor attempt to access the resource or function, the trusted service is responsible for verifying that the service requestor is permitted access to the resource or function by examining the ACL. Usually, an access control list is specific to an instance of a resource or function, or specific to a common group of resources or functions. Different resource and functions can -- and often do -- have different access control lists.

5.1.2.1.2 Describes Degree of Access Permitted to the Function or Resource

Access control lists specify the degree of access permitted to the resource. Degrees can vary between no access, to full access, and to degree in between these two extremes. Customary degrees of access

2. These configurations -- *secured, weakly secured,* and *insecure* -- were introduced in "Functional Flow" on page 25.

include: read access, write access, creation access, deletion access, administration access, and others. Some, all, or none of these degrees of access may be granted for a resource or function. It is the trusted service's responsibility to determine what level of access should be granted to its functions and resources.

5.1.2.1.3 Associates a Degree of Access To Specific Users and Services

Access control lists associate a degree of access for the resource to specific security identities and to general classes of service requestors. This means that an entry in an access control list will indicate that a user, application, or group of users has a specific set of access rights to the resource. The list contains one entry per user, service, or group. This permits the service provider to grant the differing levels of access mentioned above.

An example of such a scenario is shown in the following diagram.

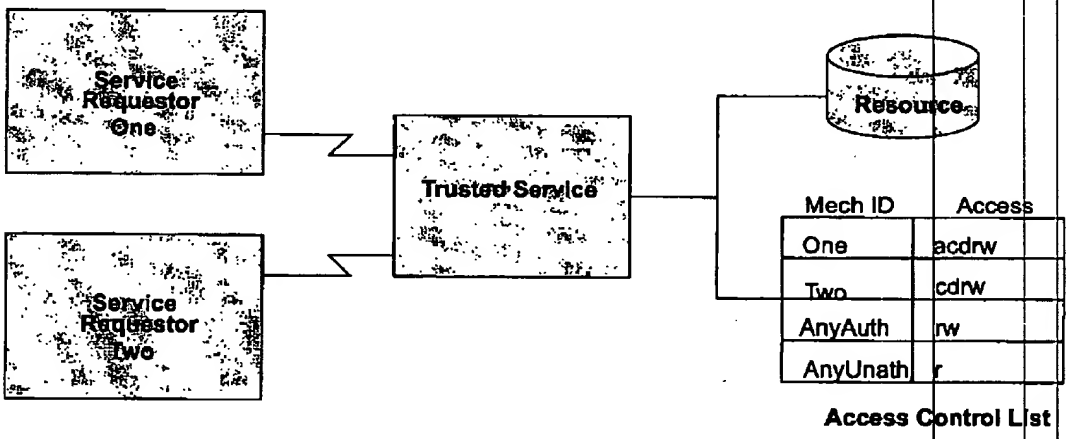

FIGURE 15 Clients requesting resources from a trusted service that uses access control to determine access

In this example, the trusted service granted service requestor One full access to the resource it controls when the trusted service set up its access controls. Service requestor Two was granted a more limited set of access, but more access than any other authentic system user or any unauthenticated system user (which has the least degree of access in this example). When service requestor Two attempts to access the resource to perform administrative functions on it, the trusted service consults the access control list to determine if this access was granted to Two. Determining that administrative access was not granted, the trusted service denies this request from Two, but would have approved such access for One.

5.1.2.2 Security Mechanism Considerations

Access controls are granted based on the security identity of the service requestor. These security identities are specific to the underlying security mechanism which is used to provide that identity.

However, the trusted service has been abstracted from any knowledge of the underlying security mechanism. It is not possible for the trusted service to know what security mechanism was used to generate the security identity of the service requestor.

If the set of security mechanisms in use by the Linux cluster should change for some reason, trusted service access control lists must be examined. System administrators must ensure that any access control lists maintained by the cluster component trusted services are updated to include any new security identities that had to be created in the new security mechanisms for existing users and services. Trusted services will be unable to perform this modification on their own.

Consider the example given in the diagram on page 50 again. If the underlying security mechanisms were modified to include a new security service called Kerberos Global Bastion (KGB), it is possible that service requestor One will need a new security identity to use with the new KGB service, perhaps ComradeOne. Should service requestor One establish its identity as ComradeOne, the trusted service will not grant it One's level of access, but instead grant it AnyAuth's level of access. In order to grant ComradeOne the level of access it deserves, the access control list used by the trusted service must be updated to include One's new security identity and to specific One's level of access to the resource.

Entries do not need to be removed from access control lists when security mechanisms are removed, although a system administrator may choose to do this to improve the overall performance of access control list entry searching.

9.0 Rationale

Reasons for choosing this design approach have been discussed in earlier sections of this document. They are summarized here for reviewing purposes.

9.1 Reduce Development and Service Investment

For all security mechanisms, the general tasks for applications and services need to perform to obtain a secured execution environment and protect resources are usually the same: agree on a convention, authenticate the parties, authorize the parties, grant or deny access, and protect information from accidental or intentional interception. The specific instructions required for each mechanism to accomplish these tasks tend to be quite different. If the security mechanisms cannot be abstracted, all applications and services must either provide multiple code paths to perform these functions based on what mechanism is in use, or must provide differing binaries that are specific to the security mechanisms.

To provide a more generic security programming model, an abstraction layer is required. The abstraction layer can handle the security mechanism specifics, while presenting generalized interfaces to applications and services to use in accomplishing the general tasks required by all security mechanisms. This reduces the development time required of all the Linux cluster software components by concentrating the mechanism specific calls in one component which the others exploit. This approach also reduces the chance of failure injection in security related software by concentrating the majority of the security mechanism interaction in one component; if all components implemented the same set of software, failures could be injected in the source code of all components. Development and service costs are expected to be reduced in this approach.

Development and service costs are also reduced in this model, because it is no longer necessary to modify trusted service or application code whenever a new security mechanism is supported on the platform. In traditional security coding, the test for configured security mechanisms would have to be extended, and a new code path would have to be inserted for the new security mechanism. By abstracting the mechanisms to a lower layer of software, applications and trusted services do not need to modify their source code in this case. Code modification is restricted to the mechanism independent layer.

9.2 Foster Development of Common Code

By removing the need for applications and trusted services to understand the differences between security mechanisms, this approach makes it possible for applications and services to use the same code path to create and use a secured execution environment. The need to detect the active security mechanism and the need to write mechanism specific code is removed, making it possible to use the same source code instructions regardless of the platform or the security mechanism in use. Common code is further fostered through the CtSec interface, which is designed around the general tasks to be performed, not the security mechanism specifics.

This approach also provides common handling of basic failure conditions applications and services would encounter in its dealings with the underlying security services. Handling of rudimentary failure cases such as invalid parameters, failed resource allocation, and so forth -- failures sometimes overlooked by applications and services that expect all resources to be available all the time -- is concentrated in the CtSec component. This helps to ensure that all basic failure cases are detected and trapped, and removes the need for applications and clients to implement checks for rudimentary failure conditions.

9.3 Allow Customer Selection of Desired Security Mechanisms

Linux clusters are expected to be introduced to existing computing installations, instead of replacing existing computing installations. Existing installations have already chosen a security mechanism and security administration model, and the Linux cluster should be designed to fit into that model instead of imposing a model on the existing installation. Since the SP represented a significant IT investment and tended to be a self-contained unit within the IT installation, SP could afford to impose its own security model on itself. Linux represents a smaller customer investment using commodity hardware and open source software, and customers expect the solution to be flexible. Therefore, the Linux cluster offering should be flexible in the security mechanisms it supports.

The initial offering of Linux cluster software from this laboratory will not completely achieve this goal, restricting the security mechanism to Kerberos version 5. However, the design is implemented to allow for easy future support of further security mechanisms, and permits the customer to select the mechanisms that will permit the Linux cluster to fit into their existing IT installation.

To allow the customer to select from multiple security mechanisms, applications and trusted services must understand and handle multiple security mechanisms. If the applications and services were required to write mechanism specific code to accomplish this, the security mechanisms that can be supported would be restricted to the set supported by the most restrictive service. In other words, if one essential trusted service supported only Kerberos 5, the only possible security mechanism to use in the cluster would be Kerberos 5, until such a point as that one trusted service is modified to support further mechanisms. This would also mean that the trusted services and applications would have to be modified to understand the new mechanism and to provide a new code path for its use.

CtSec abstracts the underlying mechanism, removing the need to modify the service and application code to support new mechanisms. Support of the mechanism is left to the CtSec mechanism abstraction layer. This permits any binary using CtSec to execute in any environment and use any security mechanism supported by CtSec, removing any constraints placed on the choice of security mechanisms from an application or trusted service point of view.

9.4 Permit Reconfiguration of Security Mechanisms

Administrators may choose to alter the security mechanisms used in their IT installation for several reasons: one mechanism has been compromised, a mechanism has been configured improperly and is

injecting failures, or a mechanism is no longer required. Not only does the security infrastructure need to permit changes in the set of security mechanisms used, but the applications and trusted services also need to support such changes as well, even if this support amounts to a restart of the software.

Applications and trusted services inherit the ability to permit security mechanism changes from the CtSec mechanism abstraction layer. Since applications and services do not directly interface with the security mechanisms, they no longer impose a restriction on changing the underlying security mechanisms. These mechanisms can now be changed, and the applications and trusted services should be capable of continuing their function.

CtSec is capable of detecting when a security mechanism is no longer in use by examining failure information from the underlying security mechanism's library interface. Instead of requiring the application or trusted service to halt and restart, CtSec informs the client that its credentials are no longer valid and must be obtained again. Properly coded applications and trusted services should already be designed to expect such an occurrence, and will handle the event in accordance with their existing credential expiration logic. Applications and trusted services are no longer required to "break out" of one path of mechanism specific code, rediscover the mechanisms currently in use, and initiate a different code path for a different security mechanism.

9.5 Facilitate Porting of SP and Cluster Software to Linux Cluster

IBM's key to success in the Linux market is to bring its current AIX based cluster software -- engineered to provide enterprise level function to a Unix based distributed operating system -- to bear on the Linux market. If the expected growth rate of the Linux market holds true to projections, Linux customers will soon find the difficulties in clustered environment that IBM has already discovered with the SP, and seek to find SP styled solutions for this market as well. To be ready, IBM needs the ability to quickly port existing SP components and AIX cluster components to the Linux cluster platform.

SP cluster software makes use of the PSSP Software Security Subsystem (S3), which sought to provide an interface independent of the underlying security mechanism. Although there are problems with providing the exact S3 infrastructure in the Linux environment[1], the security infrastructure solution should be similar in its coding model to the existing S3 interfaces.

For this reason, this design mirrors the S3 programming model to a great degree. Interface names and signatures have been modified, but the general interface flow remains the same. Components being ported to the Linux cluster should expect a minimal impact of porting to the CtSec infrastructure interfaces.

1. See "Porting of PSSP S3 Component To Linux" on page 84.

11.0 Scaling Considerations

The security infrastructure implemented by this design takes the form of a shared library. Unlike distributed subsystems that require inter-node communication in a cluster, the CtSec library interfaces with services on the local node only. Since no inter-node communication is added to the Linux cluster by the CtSec design, the addition of CtSec to the Linux cluster does not immediately impact the scalability of the Linux cluster software.

11.1 General Scalability Issues With Security

11.1.1 Authentication Scalability

CtSec places the scalability burden on the underlying security mechanisms. These mechanisms will require inter-node communication to obtain credentials and verify identities. The responsibility for ensuring that the underlying security mechanism can scale to systems the size of the proposed Linux cluster is left to the security mechanism provider. Since CtSec acts as a client of these services, CtSec cannot improve the scalability situation for them. The scalability of CtSec is directly related to the scalability of the underlying mechanisms.

Should scalability of the underlying security mechanism pose a problem, the ability of CtSec to use alternate security mechanisms can help address the problem. Should other security mechanisms prove more scalable than the original Kerberos version 5 mechanism, CtSec can be easily switched to a more scalable mechanism.

11.1.2 Authorization Scalability

Access control implementations are directly affected by the size of the Linux cluster. The more services added to the cluster, the more security identities that exist. The same holds true when more nodes are added to the Linux cluster, since one security identity will exist per service per node in the cluster. Should a trusted service use access control, ACLs can grow quite large if these access controls are based solely on identities, for the list will have to contain one entry per identity being granted (or denied) access. Obviously, larger ACLs will require more time to search, which can potentially cause timing related denial of access problems.

To address this problem, CtSec provides a security mechanism independent security identity grouping scheme to be used in access controls. By assigning common security identities to a group, access controls can be granted on a group basis. Instead of creating one entry per identity, ACLs can contain an entry per group. Using group based access should keep ACLs to a consistent size regardless of the size of the Linux cluster.

Security identity grouping is available only when LDAP is used to administer user accounts on the Linux cluster, and when all nodes in the cluster have access to a distributed LDAP repository. If this is not the case, access controls are forced to be granted (or denied) on an individual identity basis, which leads back to the original scalability problem.

System administrators need to be aware of the potential access control scaling problem when planning the Linux cluster, and include this in their decision when considering using LDAP for user account management.

11.2 Scaling Design Point for this Design

Although this solution is originally offered for the Linux cluster environment, the SP and AIX cluster design points have been used in developing the design. Fortunately, this is a relatively easy decision to make since CtSec depends on the underlying security mechanisms to scale to these environments. CtSec's implementation should not differ in small or large cluster configurations, and the currently proposed implementation should function equally well in small and large clusters.

11.3 MP Enablement

The CtSec Infrastructure library uses threads in its implementation. The library relies on the underlying operating system to efficiently schedule and dispatch these threads. Any multiprocessor capability within CtSec is directly dependent on the multiprocessor capability of the Linux operating system.

11.4 Large N-WAY RPQ Systems

This design relies on the underlying security mechanisms to scale to whatever node size is used in the Linux cluster. This applies to RPQ systems as well.

Large RPQ systems should be required to use LDAP for user account management, to permit access control scaling to large clusters. Without this, group based access control is not possible. Individual based access control will be monstrous to set up and maintain in such an environment, and the performance of access control verification will be poorer.

19.0 Coexistence

There are no specific coexistence requirements for the initial release of the Linux cluster software. In this initial release, nodes within the cluster will not be members of an alternate cluster, such as an AIX cluster. While SP nodes may comprise part or all of the Linux cluster in this release, these nodes will not be functioning as Linux cluster nodes and PSSP nodes at the same time.

Future versions of this design will address coexistence with existing AIX cluster and PSSP security infrastructures. Theoretically, it is possible for one node to use multiple security infrastructures -- such as the CtSec Infrastructure proposed in this document and PSSP's S3 infrastructure -- at the same time. However, it is unlikely that these multiple infrastructures will *inter-operate* at all, given their differing implementations and control structures. Inter-operation would also require the client application to detect in which environment it was currently executing, and invoke the correct infrastructure for that environment.

This design continues the proposal offered in reference [1]: that an instance of a trusted service or application execute in only one environment at any one time. In other words, the same server would not be serving PSSP and the Linux or AIX cluster at the same time. If a case where an SP node would reside in both a PSSP partition and a Cluster at the same time, the trusted service would provide one instance of its server for PSSP and another instance for the Cluster. Each instance could be designed to use the appropriate infrastructure for that environment.

In cases where a trusted service in one environment (PSSP) would need to become a client of a trusted service in the other environment (Linux or AIX cluster), both environments would have to shared a common security mechanism. The common security mechanism would have to manage both environments; they could not share a common mechanism but use differing users sets or security servers. If these requirements are not met, the trusted services would be forced to treat the other party as an unauthenticated user.